Денис Колисниченко

Разработка Linux-приложений

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.06
ББК 32.973.26-018.2
К60

Колисниченко Д. Н.

К60 Разработка Linux-приложений. — СПб.: БХВ-Петербург, 2012. — 432 с. — (Профессиональное программирование)

ISBN 978-5-9775-0747-9

Рассмотрены основные аспекты программирования в Linux: от программирования на языках командных оболочек bash и tcsh до создания приложений с графическим интерфейсом с использованием библиотек Tk, glib, GTK+ и средства dialog, Подробно дано программирование на C/C++ в Linux: использование компилятора gcc, ввод/вывод в Linux, создание многопоточных приложений, сетевых приложений архитектуры клиент/сервер, а также разработка модулей ядра для современной линейки ядер. Описан популярный среди разработчиков утилит язык TCL. Особое внимание уделено отладке и оптимизации программ, рассмотрены отладчик gdb и профайлер gprof.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Владимир Красовский*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Подписано в печать 30.09.11.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 34,83.
Тираж 1200 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0747-9

© Колисниченко Д. Н., 2011
© Оформление, издательство "БХВ-Петербург", 2011

Оглавление

Введение ... **11**

ЧАСТЬ I. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМАНДНОЙ ОБОЛОЧКИ ... **13**

Глава 1. Командные интерпретаторы ... **14**
1.1. Файл /etc/shells ... 14
1.2. Оболочка *sh* ... 15
1.3. Оболочка *csh* ... 16
1.4. Оболочка *ksh* ... 16
1.5. Оболочка *bash* ... 17
1.6. Оболочка *zsh* ... 17
1.7. Оболочка *tcsh* ... 18
1.8. Оболочка *ash* ... 19
1.9. Выбор оболочки ... 19

Глава 2. Командный интерпретатор *bash* ... **20**
2.1. Настройка *bash* ... 20
2.2. Автоматизация задач с помощью *bash* ... 22
2.3. Привет, мир! ... 23
2.4. Использование переменных в собственных сценариях ... 23
2.5. Передача параметров сценарию ... 25
2.6. Массивы и *bash* ... 25
2.7. Циклы ... 26
2.8. Условные операторы ... 27
2.9. Функции ... 28
2.10. Примеры сценариев ... 28
2.10.1. Сценарий мониторинга журнала ... 28
2.10.2. Переименование файлов ... 29
2.10.3. Преобразование систем счисления ... 30

Глава 3. Создание сценариев на *tcsh* ... **31**
3.1. Использование *tcsh* ... 31
3.2. Конфигурационные файлы *tcsh* ... 32

3.3. Создание сценариев на *tcsh* ... 33
3.3.1. Переменные, массивы и выражения ... 33
3.3.2. Чтение ввода пользователя ... 36
3.3.3. Переменные оболочки *tcsh* ... 36
3.3.4. Управляющие структуры ... 38
Условный оператор *if* ... 38
Условный оператор *if..then..else* ... 39
Оператор *foreach* ... 40
Оператор *while* ... 41
Оператор *switch* ... 41
3.3.5. Встроенные команды *tcsh* ... 42

Глава 4. Пакет dialog: псевдографический интерфейс пользователя ... 45
4.1. Необходимость в графическом интерфейсе ... 45
4.2. Простейшее диалоговое окно ... 46
4.3. Информационное окно ... 47
4.4. Ввод текста ... 49
4.5. Создание меню ... 51
4.6. Проблема выбора: зависимые и независимые переключатели ... 52
4.7. Выбор даты и времени ... 54
4.8. Индикатор ... 55
4.9. Диалог выбора файла ... 56
4.10. Дополнительные возможности ... 57

Глава 5. Компилятор *gcc* и вспомогательные программы ... 60
5.1. Выбор редактора ... 60
5.2. Компилятор *gcc* ... 61
5.2.1. Установка компилятора ... 61
5.2.2. Компиляция первой программы в Linux ... 62
5.2.3. Опции компилятора ... 63
5.3. Автоматическая сборка программ ... 65
5.3.1. Введение в автоматическую сборку ... 65
5.3.2. Синтаксис Makefile ... 66

ЧАСТЬ II. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА C В LINUX ... 59

Глава 6. Библиотеки. Автоматическая сборка библиотек ... 71
6.1. Динамические и статические библиотеки ... 71
6.2. Создание статической библиотеки ... 73
6.3. Создание динамической библиотеки ... 75

Глава 7. Переменные окружения ... 78
7.1. Еще один способ передачи параметров ... 78
7.2. Что такое окружение? ... 78
7.3. Чтение переменных окружения в вашей программе ... 80
7.4. Модификация окружения ... 81

Глава 8. Ввод/вывод в Linux ... 83
8.1. Понятие ввода/вывода. Перенаправление ввода/вывода в командной строке ... 83
8.2. Библиотечные функции C для организации ввода/вывода ... 85

8.3. Низкоуровневый ввод/вывод ... 89
8.3.1. Системные вызовы файлового ввода/вывода ... 89
8.3.2. Системный вызов *creat()* ... 92
8.3.3. Чтение файла: системные вызовы *open()* и *read()* ... 93
8.3.4. Системный вызов *write()* ... 95
8.3.5. Системный вызов *lseek()* ... 97

ЧАСТЬ III. СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ ... 99

Глава 9. Концепция многозадачности ... 100
9.1. Основы многозадачности Linux ... 100
9.1.1. Иерархия процессов ... 100
9.1.2. Аварийное завершение процесса ... 102
9.1.3. Программа *top*: кто больше всех расходует процессорное время ... 105
9.1.4. Команды *nice* и *renice*: изменение приоритета процесса ... 107
9.2. Функция *system()* ... 107

Глава 10. Системные вызовы для работы с процессами ... 109
10.1. Создание и запуск процессов ... 109
10.1.1. Модели описания состояний процессов ... 109
10.1.2. Особенности *fork()* ... 111
10.1.3. Семейство функций *exec* ... 113
10.2. Системный вызов *wait()*: ожидание завершения дочернего процесса ... 117
10.3. Обработка сигналов ... 119
10.4. Получение информации о процессе ... 120

Глава 11. Многопоточные приложения ... 122
11.1. Введение в потоки ... 122
11.2. Функция *pthread_create()* ... 123
11.3. Передача аргументов потоковой функции ... 126
11.4. Правильное завершение потока: функция *pthread_exit()* ... 128
11.5. Избавляемся от бесконечного цикла: функция *pthread_join()* ... 129
11.6. Получение информации о потоке ... 132
11.7. Прерывание потока ... 132

Глава 12. Взаимодействие процессов ... 133
12.1. Способы взаимодействия ... 133
12.2. Каналы ... 133
12.3. Именованные каналы типа FIFO ... 137
12.4. Очереди сообщений ... 140
12.4.1. Межпроцессное взаимодействие System V ... 140
12.4.2. Структуры ядра для работы с очередями ... 141
12.4.3. Создание очереди сообщений ... 143
12.4.4. Постановка и чтение сообщений ... 145
12.5. Семафоры ... 149
12.5.1. Введение в семафоры ... 149
12.5.2. Структуры ядра ... 149
12.5.3. Создание набора семафоров ... 150
12.5.4. Операции над семафорами ... 151
12.5.5. Управление семафором ... 152

12.6. Разделяемые сегменты памяти ... 154
12.6.1. Структуры ядра ... 154
12.6.2. Создание разделяемого сегмента памяти и привязка к нему ... 154
12.6.3. Демонстрационная программа ... 156

Глава 13. Создание модуля ядра ... 158
13.1. Что такое модуль ядра ... 158
13.2. Команды *lsmod*, *insmod*, *modprobe* ... 159
13.3. Установка необходимых пакетов ... 161
13.4. Ваш первый модуль ... 162
13.5. Компиляция модуля ... 165
13.6. Тестируем наш модуль ... 168
13.7. Сборка сложных модулей ... 172
13.8. Настоящее программирование ядра ... 172
13.8.1. Отличие обычных программ от модулей ядра ... 172
13.8.2. Пространства, пространства и еще раз пространства ... 173
13.8.3. Драйверы устройств и ядро ... 174
13.9. Символьные устройства ... 175
13.9.1. Возможные операции ... 175
13.9.2. Регистрация устройства ... 177
13.9.3. Драйвер абстрактного символьного устройства ... 177
13.10. Создание файла в /proc ... 183
13.11. Полезный пример: клавиатурный шпион ... 187

ЧАСТЬ IV. ФАЙЛОВАЯ СИСТЕМА LINUX ... 193

Глава 14. Введение в файловую систему ... 194
14.1. Родные файловые системы Linux ... 194
14.2. Особенности файловой системы Linux ... 195
14.2.1. Имена файлов в Linux ... 195
14.2.2. Файлы и устройства ... 196
14.2.3. Корневая файловая система и монтирование ... 197
14.2.4. Стандартные каталоги Linux ... 197
14.3. Внутреннее строение файловой системы ... 198
14.4. Монтирование файловых систем ... 201
14.4.1. Команды *mount* и *umount* ... 201
14.4.2. Файлы устройств и монтирование ... 202
Жесткие диски ... 202
Приводы оптических дисков ... 204
Дискеты ... 204
Флешки и USB-диски ... 204
14.4.3. Опции монтирования файловых систем ... 205
14.4.4. Монтирование разделов при загрузке ... 206
14.4.5. Подробно о UUID и файле /etc/fstab ... 208
14.4.6. Системный вызов *mount()* ... 210

Глава 15. Операции над каталогами ... 213
15.1. Команды для работы с каталогами ... 213
15.2. Функции для работы с каталогами ... 215
15.2.1. Изменение текущего каталога ... 215

15.2.2. Открываем, читаем и закрываем каталог ... 215
15.2.3. Получение информации о файлах ... 217
15.2.4. Создание и удаление каталога ... 219

Глава 16. Операции с файлами ... 220
16.1. Команды для работы с файлами ... 220
16.2. Системные вызовы для работы с файлами ... 223
16.2.1. Переименование файла: *rename()* ... 223
16.2.2. Удаление файла и каталогов: *unlink()* и *rmdir()* ... 223
16.2.3. Системный вызов *umask()* ... 224
16.2.4. Работа со ссылками ... 224

Глава 17. Получение информации о файловой системе ... 226
17.1. Список смонтированных файловых систем ... 226
17.2. Функции *basename()* и *getcwd()* ... 229

Глава 18. Права доступа к файлам и каталогам ... 230
18.1. Изменение прав доступа. Системный вызов *chmod()* ... 230
18.2. Смена владельца файла. Системный вызов *chown()* ... 233

Глава 19. Псевдофайловые системы ... 234
19.1. Что такое псевдофайловая система ... 234
19.2. Виртуальная файловая система sysfs ... 235
19.3. Виртуальная файловая система /proc ... 235
19.3.1. Информационные файлы ... 236
19.3.2. Файлы, позволяющие изменять параметры ядра ... 236
19.3.3. Файлы, изменяющие параметры сети ... 237
19.3.4. Файлы, изменяющие параметры виртуальной памяти ... 238
19.3.5. Файлы, позволяющие изменить параметры файловых систем ... 238
19.3.6. Как сохранить изменения ... 238

ЧАСТЬ V. СЕТЕВОЕ ПРОГРАММИРОВАНИЕ ... 241

Глава 20. Введение в TCP/IP ... 242
20.1. Модель OSI ... 242
20.2. Что такое протокол ... 244
20.3. Адресация компьютеров ... 245

Глава 21. Программирование сокетов: теория ... 249
21.1. Что такое сокет ... 249
21.2. Создание и связывание сокета ... 250
21.3. Установление связи с удаленным компьютером ... 252
21.4. Передача данных ... 254
21.5. Завершение сеанса связи ... 255

Глава 22. Программирование сокетов: практика ... 256
22.1. Создание приложения клиент/сервер ... 256
22.1.1. Программа-сервер ... 256
22.1.2. Программа-клиент ... 259
22.2. Параметры сокета ... 260

22.3. Сигналы, связанные с сокетами ... 263
22.4. Неблокирующие операции ... 264

ЧАСТЬ VI. СОЗДАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА СРЕДСТВАМИ TCL/TK ... 265

Глава 23. Введение в TCL/Tk ... 266
23.1. Знакомство с TCL ... 266
23.2. Установка TCL/Tk ... 266
23.3. Первая программа ... 267

Глава 24. Синтаксис TCL ... 270
24.1. Знакомство с синтаксисом TCL ... 270
24.1.1. Формат TCL-сценария ... 270
24.1.2. Команды *puts* и *format*: вывод и форматирование строки ... 271
24.1.3. Группировка аргументов ... 272
24.1.4. Переменные ... 272
24.1.5. Процедуры ... 274
24.1.6. Получаем ввод пользователя ... 274
24.1.7. Математические операции ... 275
24.1.8. Условная команда *if* ... 276
24.1.9. Команда *while* ... 277
24.1.10. Команда *for* ... 277
24.2. Строки ... 277
24.2.1. Команда *string* ... 277
24.2.2. Сравнение строк ... 279
24.2.3. Получаем информацию о строках ... 280
24.2.4. Модификация строк ... 282
24.2.5. Конкатенация строк ... 283
24.3. Списки ... 283
24.3.1. Команда *list*: создание списка ... 283
24.3.2. Команда *concat*: слияние списков ... 284
24.3.3. Команда *lappend*: добавление элемента в конец списка ... 284
24.3.4. Доступ к элементам списка ... 284
24.3.5. Вставка новых элементов ... 285
24.3.6. Замена и удаление элементов списка ... 285
24.3.7. Поиск элемента ... 285
24.3.8. Сортировка списка ... 287
24.3.9. Преобразование строки в список и обратно ... 287
24.3.10. Цикл *foreach* ... 288
24.4. Массивы ... 289
24.4.1. Отличие массивов от списков ... 289
24.4.2. Команда *array*: обработка массивов ... 289
24.5. Ошибки начинающих TCL-программистов ... 290

Глава 25. Работа с файлами ... 292
25.1. Открываем и закрываем файлы ... 292
25.2. Чтение файла ... 293
25.3. Запись файлов ... 295
25.4. Произвольный доступ к файлу ... 296

Глава 26. Понятие о виджетах ... 297
26.1. Tk-программирование ... 297
26.2. Компоненты Tk-приложения и имена виджетов ... 300

Глава 27. Основные элементы графического интерфейса ... 302
27.1. Команда *pack* ... 302
27.2. Команда *button* ... 304
27.3. Команда *checkbutton* ... 307
27.4. Зависимые переключатели ... 311
27.5. Создание меню ... 313
27.6. Поля ввода ... 315
27.7. Списки и домашнее задание ... 318
27.8. Программирование событий. Команда *bind* ... 319

Глава 28. Многооконный интерфейс ... 321
28.1. Менеджер геометрии *grid* ... 321
28.1.1. Относительное размещение ... 321
28.1.2. Абсолютное размещение ... 323
28.1.3. Объединение ячеек ... 325
28.2. Фреймы ... 326
28.3. Создание окон ... 327
28.4. Сообщения ... 328
28.5. Диалоги открытия и сохранения файла ... 330

Глава 29. Практический пример ... 332
29.1. Постановка задачи ... 332
29.2. Создание оболочки ... 333
29.3. Запуск оболочки ... 336
29.4. Последние штрихи ... 336

ЧАСТЬ VII. БИБЛИОТЕКА GTK+ ... 337

Глава 30. Знакомство с библиотекой ... 338
30.1. Введение в GTK+ ... 338
30.2. Библиотека GLib ... 339
30.2.1. Типы данных ... 339
30.2.2. Строки в GLib ... 340
30.2.3. Функции распределения памяти ... 341
30.2.4. Списки ... 342
30.2.5. Использование таймеров ... 344

Глава 31. Первая программа на GTK+ ... 346
31.1. Виджеты, контейнеры, сигналы и события ... 346
31.2. Создание первой программы ... 347
31.3. Компиляция программы ... 348
31.4. Совершенствование программы. Обработчик сигнала ... 351

Глава 32. Виджеты ... 354
32.1. Подробно о сигналах ... 354
32.1.1. Сигналы и события ... 354
32.1.2. Виджет *EventBox* ... 356

32.2. Русский текст и GTK ... 360
32.3. Состояния виджета ... 361
32.4. Контейнеры, поля ввода и кнопки ... 362
32.5. Зависимые и независимые переключатели ... 370
32.6. Список *CList* ... 375
32.7. Диалог выбора файлов ... 379
32.8. Визуальная разработка интерфейса пользователя ... 381

Глава 33. Редактор интерфейсов Glade ... 382
33.1. Быстрая разработка приложений ... 382
33.2. Установка Glade ... 383
33.3. Использование Glade ... 384
33.4. Создание программы ... 388
33.5. Компиляция программы ... 390
33.6. Рекомендуемая литература ... 390

ЧАСТЬ VIII. ОТЛАДКА И ОПТИМИЗАЦИЯ ПРОГРАММЫ ... 393

Глава 34. Отладка программ. Трассировка системных вызовов ... 394
34.1. Для чего нужна отладка программ ... 394
34.2. Введение в отладчик *gdb* ... 396
34.3. Пример использования *gdb* ... 399
34.4. Трассировка системных вызовов ... 404

Глава 35. Оптимизация программы ... 407
35.1. Назначение и основные опции профайлера *gprof* ... 407
35.2. Практическое использование профайлера ... 408

Заключение ... 413

Приложение. Ядро Linux ... 415
П1. Установка исходных кодов ядра ... 415
П2. Настройка ядра ... 417
П3. Компиляция ядра ... 419

Предметный указатель ... 423

Введение

Процесс разработки приложений для Linux очень многогранный. Вы можете заниматься как разработкой сценариев командной оболочки, так и разработкой модулей ядра. При создании сложных проектов вполне возможно, что оба эти подхода будут использоваться в одном проекте. Все зависит от поставленных целей.

Книга охватывает основные моменты программирования в Linux — от создания простых сценариев на языках оболочек `bash` и `tcsh` до создания программ на C/C++ с графическим интерфейсом.

Предполагается, что читатель уже знаком с языками C и C++: эта книга не учебник по C/C++, в ней вы не найдете описания синтаксиса и стандартных функций этих языков программирования. Зато в ней рассмотрены нюансы создания программ на этих языках в Linux. Поверьте, хотя синтаксис языка и набор стандартных функций остается тем же, таких нюансов достаточно много: у каждой операционной системы есть свои особенности, и если вы раньше программировали в Windows, то будете удивлены, когда узнаете, что в Linux "все не так" (ну, или почти все).

Книга разделена на восемь частей. В *части I* рассматривается синтаксис встроенных языков оболочек `bash` и `tcsh`. Также в этой части будет рассмотрен пакет dialog, позволяющий создавать интерфейс пользователя для сценариев.

Знать синтаксис оболочки нужно каждому уважающему себя программисту. По сути, знание синтаксиса можно приравнять к знанию самой оболочки: не знаете, как программировать в `bash`, значит, не знаете, как использовать эту оболочку. Да и не всегда нужно создавать программу на C: для некоторых простых (и не очень) задач это нецелесообразно. Как вы думаете, почему сценарии системы инициализации `init` написаны на языке командной оболочки, а не на C? Ведь можно было бы написать все эти программы на C, тогда бы сократилось время загрузки системы — ведь скомпилированные программы будут выполняться быстрее. Но тогда, чтобы внести малейшее изменение в процесс загрузки (поверьте, в ранних версиях дистрибутивов это приходилось делать чаще, чем можно представить), нужно перекомпилировать программу. А в случае со сценарием достаточно открыть его в редакторе и изменить код, после чего сценарий "готов к употреблению". В свою очередь это также освобождает пользователя от знания синтаксиса C и от необходимости установки компилятора на каждый компьютер.

Да, были системы инициализации, полностью написанные на C, например `init-ng` (Init Next Generation), но что бы вы думали? Они не прижились...

Часть II посвящена основам программирования на C в Linux. Вы познакомитесь с компилятором `gcc`, утилитой автоматической сборки программ `make`, с переменными окружения, также будет рассмотрен ввод/вывод в Linux. Так сказать, необходимый минимум для начала настоящего программирования в Linux. Прочитав эту часть книги, вы сможете перенести в Linux ваши программы, разработанные с использованием стандартных библиотек C. Все, что нужно для этого — знать, как скомпилировать программу в Linux, а об этом как раз и говорится в *части II*.

В *части III* рассматривается системное программирование в Linux: организация межпроцессного взаимодействия (IPC), потоки, создание модулей ядра и т. д. Все это позволит почувствовать себя настоящим программистом, но ради справедливости нужно отметить, что далеко не всем программистам нужны средства, описанные в этой части книги. Если вы планируете написать текстовый редактор, то нет особого смысла рассматривать программирование ядра.

Файловая система Linux заслуживает отдельного разговора и такой разговор будет — в *части IV*. Сначала будет рассмотрена файловая система глазами пользователя, а затем — программиста. Понимаю, что книга может попасть в руки программиста, ни разу в жизни не видевшего Linux, и в этом случае нет смысла рассматривать системные вызовы для работы с файлами и каталогами, если человек не знает, как скопировать или удалить файл в командной оболочке.

В *части V* рассматривается сетевое программирование. Мы создадим собственный сервер и собственный клиент — две программы, которые могут обмениваться данными по сети. Вам же останется только разработать протокол — набор правил для обмена информацией, и у вас будет полноценное приложение архитектуры клиент/сервер.

Части VI и *VII* посвящены созданию графического интерфейса пользователя. Сначала будет рассмотрен язык программирования TCL (наверное, в Windows вы о таком и не слышали) и графическая библиотека Tk, используемая в паре с TCL. Затем (в *части VII*) будут рассмотрены библиотеки GLib и GTK+ для создания GUI. Также будет рассмотрен редактор интерфейсов Glade, позволяющий за несколько минут создать интерфейс небольшого окна и в результате этого существенно сократить код GTK-программы.

И наконец, *часть VIII* посвящена отладке и оптимизации программы. Будут рассмотрены отладчик `gdb` и профайлер `gprof`.

В *приложении* вы найдете инструкции по перекомпиляции ядра — они вам понадобятся на случай, если вы будете создавать модули ядра.

Вот теперь самое время приступить к чтению книги. О том, как связаться со мной, вы сможете прочитать в *заключении*.

ЧАСТЬ I

Программирование на языке командной оболочки

Глава 1.	Командные интерпретаторы
Глава 2.	Командный интерпретатор *bash*
Глава 3.	Создание сценариев на *tcsh*
Глава 4.	Пакет dialog: псевдографический интерфейс пользователя

Первая часть книги посвящена программированию на языке командного интерпретатора Linux. Кому нужны такие программы? Не всегда целесообразно писать программу на языке C. Некоторые операции можно с легкостью запрограммировать, используя язык командного интерпретатора. А если вы еще освоите дополнительные средства, например язык awk, используемый для обработки текстовых файлов по определенному шаблону, то написание некоторых программ займет у вас всего несколько минут. А вот в C придется изобретать велосипед заново...

ГЛАВА 1

Командные интерпретаторы

1.1. Файл /etc/shells

У командных интерпретаторов есть свои преимущества в плане переносимости. К примеру, пусть у вас будет 64-битная система, вы скомпилировали программу, написанную на C. Чтобы она заработала на 32-битной системе или же вообще на компьютере с другой архитектурой (ведь мир не сошелся клином на x86 и x86-64), то программу придется перекомпилировать. В случае со сценарием командной оболочки этого делать не требуется — просто скопируйте сценарий на другую систему и выполните его. Ведь сценарий — это обычный файл, поэтому проблем с переносимостью у вас не будет.

Конечно, сценарии командного интерпретатора не панацея. Сложных программ на них не напишешь. Я имею в виду действительно сложных, а не больших! Большую программу (свыше 1000 строк) написать на языке командного интерпретатора можно, пример тому — сценарии инициализации Linux, которые в некоторых дистрибутивах длиннее, чем лимузин Билла Гейтса. Что же касается больших программ, то выполняться они будут значительно медленнее, чем их аналоги, написанные на C. Ведь выполнением программы занимается командный интерпретатор, а для выполнения скомпилированной C-программы не нужны какие-либо вспомогательные программы (если, конечно, вы их не вызываете из своей программы).

Также к недостаткам сценариев командной оболочки можно отнести необходимость установки той командной оболочки, на языке которой написан сценарий. Это небольшой недостаток, т. к. обычно сценарии пишутся на `bash` или `tcsh`, а эти командные оболочки почти всегда установлены в Linux. Что же касается FreeBSD, то в ней `bash` по умолчанию не установлен, поэтому его придется установить самостоятельно. Впрочем, о выборе командного интерпретатора для написания сценариев мы и поговорим в данной главе.

По умолчанию во всех современных дистрибутивах используется командный интерпретатор `bash`. Основное предназначение `bash`, как и любой другой оболочки, — выполнение команд, введенных пользователем. Пользователь вводит команду, `bash` ищет программу, соответствующую команде, в каталогах, указанных в переменной

окружения PATH. Если такая программа найдена, то bash запускает ее и передает введенные пользователем параметры. В противном случае выводится сообщение о невозможности выполнения команды.

Кроме bash существуют и другие оболочки — sh, csh, ksh, zsh и пр. Все командные оболочки, установленные в системе, прописаны в файле /etc/shells. Список оболочек может быть довольно длинным. В листинге 1.1 представлен файл /etc/shells дистрибутива Fedora (установка по умолчанию).

Листинг 1.1. Файл /etc/shells дистрибутива Fedora

```
/bin/ash
/bin/bash
/bin/csh
/bin/false
/bin/ksh
/bin/sh
/bin/tcsh
/bin/true
/bin/zsh
/usr/bin/csh
/usr/bin/ksh
/usr/bin/bash
/usr/bin/tcsh
/usr/bin/zsh
```

С точки зрения пользователя указанные оболочки мало чем отличаются. Все они позволяют выполнять введенные пользователем команды. Но оболочки используются не только для выполнения команд, а еще и для автоматизации задач с помощью *сценариев*. Так вот, все эти оболочки отличаются синтаксисом языка описания сценариев.

Примечание

В листинге 1.1 программы /bin/false и /bin/true не являются оболочками. Это "заглушки", которые можно использовать, если вы хотите отключить ту или иную учетную запись пользователя. При входе пользователя в систему запускается установленная для него оболочка. Для каждого пользователя имеется возможность задать свою оболочку (изменить оболочку пользователь может самостоятельно командой chsh). Так вот, если для пользователя задать оболочку /bin/false (или /bin/true), он не сможет войти в систему. Точнее, он войдет в систему, но и сразу выйдет из нее, поскольку обе "заглушки" ничего не делают, а просто возвращают значение 0 (для false) или 1 (для true). Сессия же пользователя длится до завершения работы его оболочки.

1.2. Оболочка *sh*

Самым первым командным интерпретатором (оболочкой) в операционной системе UNIX (да, именно UNIX, поскольку корни Linux уходят в далекие 70-е годы прошлого столетия) была sh (сокращение от *shell*). Данная оболочка до сих пор используется в современных версиях Linux (и FreeBSD).

Оболочка sh была разработана Стивеном Борном (Steve Bourne), поэтому ее второе название — Bourne Shell. Изначально sh была создана для операционной системы AT&T (разработка Bell Labs). Чуть позже sh была усовершенствована и вошла в состав POSIX (Portable Operating System Interface for UNIX — Переносимый интерфейс операционных систем UNIX). Усовершенствованная версия sh до сих пор устанавливается (но не используется по умолчанию) в современных версиях FreeBSD.

С точки зрения пользователя оболочка sh не очень удобна, поэтому пользователи предпочитают другие оболочки, например tcsh или bash.

1.3. Оболочка *csh*

Оболочка csh (C Shell) по умолчанию используется в FreeBSD. Разработка csh началась еще в первых версиях BSD (Linux будет создан лет через 15). Тогда в институте Беркли начали создавать новую оболочку (csh), потому что не захотели мириться с ограничениями sh.

Внутренний синтаксис csh очень напоминает язык программирования C, поэтому он должен был понравиться программистам (а в то время все пользователи компьютеров являлись программистами). Хотя сами программисты отмечали, что синтаксис не очень удобен, даже несмотря на то, что он похож на C.

По сравнению с sh, у csh есть множество преимуществ: она умеет управлять заданиями, хранит историю ранее введенных команд, а также у csh есть сценарии, которые выполняются при входе пользователя (запуске оболочки) и при выходе пользователя (когда пользователь вводит команду exit). В то время у sh не было таких сценариев, которые оказались очень удобными.

С точки зрения обычного использования оболочки (а не программирования) csh тоже была на высоте.

В последних версиях FreeBSD и Linux вместо csh используется ее усовершенствованная версия — tcsh, а файл /bin/csh — это просто ссылка на /bin/tcsh.

1.4. Оболочка *ksh*

Не хочется делать экскурс в историю UNIX, но пару слов сказать все же придется. Изначально система UNIX появилась в лабораториях компании AT&T, позже появились версии UNIX института Беркли (операционная система называлась BSD). Так уж сложилось исторически, что AT&T и институт Беркли постоянно конкурировали между собой. Как только в Беркли разработали оболочку csh, в AT&T принялись разрабатывать собственную оболочку, которая получила название ksh (Korn Shell) — по имени разработчика Дэвида Корна (David Korn).

Оболочка ksh по функциям похожа на csh: есть поддержка управления заданиями, история команд, позволяет назначать командам псевдонимы, а также создавать конфигурационные файлы для подоболочек.

Несмотря на то что оболочка была разработана в 1986 году, она до сих пор используется в некоторых версиях UNIX по умолчанию, а также устанавливается по умолчанию во всех дистрибутивах Linux (но не используется по умолчанию). Правда, изначально ksh — это коммерческий продукт, поэтому в FreeBSD и Linux используется не ksh, а ее бесплатная версия — pdksh, но для краткости исполнимый файл называется ksh.

Начинающим пользователям ksh не понравится (лучше использовать bash) — она слишком неудобна в использовании, зато у нее довольно развитый синтаксис внутреннего языка, что понравится программистам.

1.5. Оболочка *bash*

Командный интерпретатор bash (Bourne Again Shell) был разработан Фондом свободного программного обеспечения (Free Software Foundation, FSF). За основу была взята оболочка sh. Оболочка стала очень популярной и сейчас используется по умолчанию во всех дистрибутивах Linux.

Оболочка bash может использоваться также и для запуска сценариев sh, поэтому sh во многих системах уже не устанавливается, а файл /bin/sh — это ссылка на /bin/bash.

С точки зрения пользователей оболочка bash намного удобнее, чем ksh. Вы можете легко редактировать командную строку, просматривать историю команд, создавать псевдонимы команд, создавать переменные окружения и использовать их в собственных сценариях. Как и в csh, в bash есть сценарии, которые вызываются при запуске оболочки и при выходе из нее.

Синтаксис bash довольно прост, поэтому бóльшая часть сценариев, разрабатываемых в Linux, пишется именно на bash.

1.6. Оболочка *zsh*

Сейчас мы поближе познакомимся с оболочкой zsh, которая становится все более популярной.

До того как я познакомился с zsh, я считал самой удобной оболочку bash. Однако это не так.

Что же удобного в zsh? Во-первых, навигация. В bash для перехода в каталог /dir/subdir1/subdir2 нужно ввести команду:

```
cd /dir/subdir1/subdir2
```

Можно использовать автодополнение bash — вводить начальные символы каталога и нажимать клавишу <Tab>. Это будет выглядеть примерно так:

```
cd /dir/sub [Tab]/subdi [Tab]
```

В zsh можно ввести:

```
/d/s/s
```

Затем нажать клавишу <Tab> — вы перейдете в нужный каталог. Например, для перехода в /etc/sysconfig/network нужно ввести /e/s/n и нажать клавишу <Tab>. Кстати, команда cd уже не нужна.

Покажу еще один трюк. Предположим, у нас есть каталог files, а в нем есть каталоги f1 и f2. Внутри каждого каталога f* есть каталоги sources и last. То есть структура каталогов будет примерно такой:

```
/files/f1/sources/last
/files/f2/sources/last
```

Пусть мы находимся в каталоге /files/f1/sources/last. Для перехода в каталог /files/f2/sources/last введите команду:

```
cd 1 2
```

Но одной лишь навигацией возможности zsh не ограничиваются. Можно, например, использовать вот такое перенаправление:

```
< /var/log/messages
```

Оболочка запустит программу, указанную в переменной $PAGER. В большинстве случаев это аналогично команде:

```
cat /var/log/messages | less
```

Все возможности zsh в этой главе мы рассматривать не будем — их намного больше, чем вам кажется. Если вы заинтересовались, то прочитайте следующие страницы:

- **http://opennet.ru/base/dev/zsh_intro.txt.html**;
- **http://citkit.ru/articles/1083/**;
- **http://alexott.net/ru/writings/zsh/index.html**;
- **http://habrahabr.ru/blogs/linux/82537/**.

1.7. Оболочка *tcsh*

Оболочка tcsh является модифицированной версией csh. Буква t в названии означает TENEX: изначально оболочка была разработана для операционной системы TENEX (использовалась в далеком прошлом на компьютерах DEC PDP-10).

В tcsh усовершенствована функция редактирования командной строки, есть автозавершение команд (как в bash). Кроме того, tcsh может распознавать потенциально опасные команды. Если вы от имени root попытаетесь удалить все файлы, оболочка потребует подтверждения.

Оболочка tcsh очень удобна в использовании, но ее синтаксис сценариев сложнее, чем у bash. Однако далее мы все же рассмотрим разработку сценариев на tcsh, чтобы вы смогли оценить сложность создания разработки сценариев на bash и на tcsh.

1.8. Оболочка *ash*

Оболочка `ash` (Almquist shell) — самая простая командная оболочка. Это самая маленькая оболочка, доступная для UNIX (у нее самые низкие требования к дисковому пространству).

У `ash` всего 24 встроенных команды и 10 опций командной строки. Обычно `ash` используется при загрузке Linux в однопользовательском режиме (или в режиме восстановления).

Оболочка `ash` совместима с `sh`, с ее помощью можно проверить сценарии на совместимость с традиционным синтаксисом `sh`. А в операционной системе NetBSD оболочка `ash` используется вместо /bin/sh.

1.9. Выбор оболочки

Какую оболочку выбрать? Первым делом нужно оценить простоту использования оболочки. Ведь вы будете использовать эту оболочку каждый день, поэтому простота использования должна быть на первом месте.

Затем нужно оценить простоту синтаксиса оболочки. Конечно, это только в том случае, если вы планируете разрабатывать собственные сценарии. Также не нужно забывать, что вы можете использовать одну оболочку, а разрабатывать сценарии — на языке другой оболочки. Например, в повседневной работе вы можете использовать `zsh`, а разрабатывать сценарии на языке `bash`.

Довольно удобны в использовании оболочки `bash`, `tcsh` и `zsh`. Скорее всего, вы выберете одну из них. А вот для программирования вы будете использовать или `bash`, или `tcsh` (синтаксис `zsh` не очень понятен). Именно эти две оболочки будут рассмотрены в последующих двух главах.

ГЛАВА 2

Командный интерпретатор *bash*

2.1. Настройка *bash*

`bash` — это самая популярная командная оболочка (командный интерпретатор) Linux. Основное предназначение `bash` — выполнение команд, введенных пользователем. Пользователь вводит команду, `bash` ищет программу, соответствующую команде, в каталогах, указанных в переменной окружения `PATH`. Если такая программа найдена, то `bash` запускает ее и передает ей введенные пользователем параметры. В противном случае выводится сообщение о невозможности выполнения команды.

Файл /etc/profile содержит глобальные настройки `bash`, он влияет на всю систему — на каждую запущенную оболочку. Обычно /etc/profile не нуждается в изменении, а при необходимости изменить параметры `bash` редактируют один из файлов:

- ~/.bash_profile — обрабатывается при каждом входе в систему;
- ~/.bashrc — обрабатывается при каждом запуске дочерней оболочки;
- ~/.bash_logout — обрабатывается при выходе из системы.

Файл ~/.bash_profile часто не существует, а если и существует, то в нем есть всего одна строка:

```
source ~/.bashrc
```

Данная строка означает, что нужно прочитать файл .bashrc. Поэтому будем считать основным конфигурационным файлом файл .bashrc. Но помните, что он влияет на оболочку текущего пользователя (такой файл находится в домашнем каталоге каждого пользователя, не забываем: "~" означает домашний каталог). Если же вдруг понадобится задать параметры, которые повлияют на всех пользователей, то нужно редактировать файл /etc/profile.

В файле .bash_history (тоже находится в домашнем каталоге) хранится история команд, введенных пользователем. Так что вы можете просмотреть свои же команды, которые накануне вводили.

Какие настройки могут быть в .bashrc? Как правило, в этом файле задаются псевдонимы команд, определяется внешний вид приглашения командной строки, задаются значения переменных окружения.

Псевдонимы команд задаются с помощью команды `alias`, вот несколько примеров:

```
alias h='fc -l'
alias ll='ls -laFo'
alias l='ls -l'
alias g='egrep -i'
```

Псевдонимы работают просто: при вводе команды `l` на самом деле будет выполнена команда `ls -l`.

Теперь рассмотрим пример изменения приглашения командной строки. Глобальная переменная `PS1` отвечает за внешний вид командной строки. По умолчанию командная строка имеет формат:

```
пользователь@компьютер:рабочий_каталог
```

Значение `PS1` при этом будет:

```
PS1='\u@\h:\w$'
```

В табл. 2.1 приведены допустимые модификаторы командной строки.

Таблица 2.1. *Модификаторы командной строки*

Модификатор	Описание
`\a`	ASCII-символ звонка (код 07). Не рекомендуется его использовать — очень скоро начнет раздражать
`\d`	Дата в формате "день недели, месяц, число"
`\h`	Имя компьютера до первой точки
`\H`	Полное имя компьютера
`\j`	Количество задач, запущенных в оболочке в данное время
`\l`	Название терминала
`\n`	Символ новой строки
`\r`	Возврат каретки
`\s`	Название оболочки
`\t`	Время в 24-часовом формате (ЧЧ:ММ:СС)
`\T`	Время в 12-часовом формате (ЧЧ:ММ:СС)
`\@`	Время в 12-часовом формате (AM/PM)
`\u`	Имя пользователя
`\v`	Версия `bash` (сокращенный вариант)
`\V`	Версия `bash` (полная версия: номер релиза, номер патча)
`\w`	Текущий каталог (полный путь)

Таблица 2.1 *(окончание)*

Модификатор	Описание
`\W`	Текущий каталог (только название каталога, без пути)
`\!`	Номер команды в истории
`\#`	Системный номер команды
`\$`	Если UID пользователя равен 0, будет выведен символ `#`, иначе — символ `$`
`\\`	Обратный слэш
`$ ( )`	Подстановка внешней команды

Вот пример альтернативного приглашения командной строки:

```
PS1='[\u@\h] $(date +%m/%d/%y) \$'
```

Вид приглашения будет такой:

```
[denis@host] 12/06/10 $
```

В квадратных скобках выводится имя пользователя и имя компьютера, затем используется конструкция `$()` для подстановки даты в нужном нам формате и символ приглашения, который изменяется в зависимости от идентификатора пользователя.

Установить переменную окружения можно с помощью команды `export`, что будет показано позже.

2.2. Автоматизация задач с помощью *bash*

Представим, что нам предстоит выполнить резервное копирование всех важных файлов, для чего нужно создать архивы каталогов /etc, /home и /usr. Понятно, что понадобятся три команды вида:

```
tar -cvjf имя_архива.tar.bz2 каталог
```

Затем нам нужно записать все эти три файла на DVD с помощью любой программы для прожига DVD.

Если выполнять данную операцию раз в месяц (или хотя бы раз в неделю), то ничего страшного. Но представьте, что вам нужно делать это каждый день или даже несколько раз в день? Думаю, такая рутинная работа вам быстро надоест. А ведь можно написать *сценарий*, который сам будет создавать резервные копии и записывать их на DVD! Все, что вам нужно, — это вставить чистый DVD перед запуском сценария.

Можно пойти и иным путем. Написать сценарий, который будет делать резервные копии системных каталогов и записывать их на другой раздел жесткого диска. Ведь не секрет, что резервные копии делаются не только на случай сбоя системы, но и для защиты от некорректного изменения данных пользователем. Помню, удалил важную тему форума и попросил своего хостинг-провайдера сделать откат. Я был приятно удивлен, когда мне предоставили на выбор три резервные копии — оста-

лось лишь выбрать наиболее подходящую. Не думаете же вы, что администраторы провайдера только и занимались тем, что три раза в день копировали домашние каталоги пользователей? Поэтому автоматизация — штука полезная, и любому администратору нужно знать, как автоматизировать свою рутинную работу.

2.3. Привет, мир!

По традиции напишем первый сценарий, выводящий всем известную фразу: "Привет, мир!" (Hello, world!). Для редактирования сценариев вы можете использовать любимый текстовый редактор, например `nano` или `ee` (листинг 2.1).

Листинг 2.1. Первый сценарий

```
#!/bin/bash
echo "Привет, мир!"
```

Первая строка нашего сценария — это указание, что он должен быть обработан программой /bin/bash. Обратите внимание: если между `#` и `!` окажется пробел, то данная директива не сработает, поскольку будет воспринята как обычный комментарий. Комментарии начинаются, как вы уже догадались, с символа решетки:

```
# Комментарий
```

Вторая строка — это оператор `echo`, выводящий нашу строку. Сохраните сценарий под именем hello и введите команду:

```
$ chmod +x hello
```

Для запуска сценария введите команду:

```
./hello
```

На экране вы увидите строку:

```
Привет, мир!
```

Чтобы вводить для запуска сценария просто `hello` (без `./`), сценарий нужно скопировать в каталог /usr/bin (точнее, в любой каталог из переменной окружения `PATH`):

```
# cp ./hello /usr/bin
```

2.4. Использование переменных в собственных сценариях

В любом серьезном сценарии вы не обойдетесь без использования *переменных*. Переменные можно объявлять в любом месте сценария, но до места их первого применения. Рекомендуется объявлять переменные в самом начале сценария, чтобы потом не искать, где вы объявили ту или иную переменную.

Для объявления переменной используется следующая конструкция:

```
переменная=значение
```

Пример объявления переменной:

```
ADDRESS=www.dkws.org.ua
echo $ADDRESS
```

Обратите внимание на следующие моменты:

- при объявлении переменной знак доллара не ставится, но он обязателен при использовании переменной;
- при объявлении переменной не должно быть пробелов до и после знака =.

Значение для переменной указывать вручную не обязательно — его можно прочитать с клавиатуры:

```
read ADDRESS
```

или со стандартного вывода программы:

```
ADDRESS='hostname'
```

Чтение значения переменной с клавиатуры осуществляется с помощью инструкции `read`. При этом указывать символ доллара не нужно. Вторая команда устанавливает в качестве значения переменной `ADDRESS` вывод команды `hostname`.

В Linux часто используются *переменные окружения*. Это специальные переменные, содержащие служебные данные. Вот примеры некоторых часто используемых переменных окружения:

- `BASH` — полный путь до исполняемого файла командной оболочки `bash`;
- `BASH_VERSION` — версия `bash`;
- `HOME` — домашний каталог пользователя, который запустил сценарий;
- `HOSTNAME` — имя компьютера;
- `RANDOM` — случайное число в диапазоне от 0 до 32 767;
- `OSTYPE` — тип операционной системы;
- `PWD` — текущий каталог;
- `PS1` — строка приглашения;
- `UID` — ID пользователя, который запустил сценарий;
- `USER` — имя пользователя.

Для установки собственной переменной окружения используется команда `export`:

```
# Присваиваем переменной значение
$ADDRESS=ww.dkws.org.ua
# Экспортируем переменную – делаем ее переменной окружения
# После этого переменная ADDRESS будет доступна в других сценариях
export $ADDRESS
```

2.5. Передача параметров сценарию

Очень часто сценариям нужно передавать различные параметры, например режим работы или имя файла/каталога. Для передачи параметров используются следующие специальные переменные:

- `$0` — содержит имя сценария;
- `$n` — содержит значение параметра (`n` — номер параметра);
- `$#` — позволяет узнать количество параметров, которые были переданы.

Рассмотрим небольшой пример обработки параметров сценария. Конструкцию `case-esac` мы еще не рассматривали, но общий принцип должен быть понятен (листинг 2.2).

Листинг 2.2. Пример обработки параметров сценария

```
# Сценарий должен вызываться так:
# имя_сценария параметр

# Анализируем первый параметр
case "$1" in
  start)
        # Действия при получении параметра start
        echo "Запускаем сетевой сервис"
        ;;
  stop)
        # Действия при получении параметра stop
        echo "Останавливаем сетевой сервис"
        ;;
*)
        # Действия в остальных случаях
        # Выводим подсказку о том, как нужно использовать сценарий, и
        # завершаем работу сценария
echo "Usage: $0 {start|stop }"
        exit 1
        ;;
esac
```

Думаю, приведенных комментариев достаточно, поэтому подробно рассматривать работу сценария из листинга 2.2 не будем.

2.6. Массивы и *bash*

Интерпретатор `bash` позволяет использовать *массивы*. Массивы объявляются подобно переменным.

Вот пример объявления массива:

```
ARRAY[0]=1
ARRAY[1]=2

echo $ARRAY[0]
```

2.7. Циклы

Как и в любом языке программирования, в bash можно использовать *циклы*. Мы рассмотрим циклы for и while, хотя вообще в bash доступны также циклы until и select, но они применяются довольно редко.

Синтаксис цикла for выглядит так:

```
for переменная in список
do
команды
done
```

В цикле при каждой итерации переменной будет присвоен очередной элемент списка, над которым будут выполнены указанные команды. Чтобы было понятнее, рассмотрим небольшой пример:

```
for n in 1 2 3;
do
echo $n;
done
```

Обратите внимание: список значений и список команд должны заканчиваться точкой с запятой.

Как и следовало ожидать, наш сценарий выведет на экран следующее:

```
1
2
3
```

Синтаксис цикла while выглядит немного иначе:

```
while условие
do
команды
done
```

Цикл while выполняется до тех пор, пока истинно заданное условие. Подробно об условиях мы поговорим в следующем разделе, а сейчас напишем аналог предыдущего цикла, т. е. нам нужно вывести 1, 2 и 3, но с помощью while, а не for:

```
n=1
while [ $n -lt 4 ]
do
  echo "$n "
  n=$(( $n+1 ));
done
```

2.8. Условные операторы

В bash доступно два *условных оператора*, if и case. Синтаксис оператора if следующий:

```
if условие_1 then
       команды_1
elif условие_2 then
       команды_2
...
elif условие_N then
       команды_N
else
       команды_N+1
fi
```

Оператор if в bash работает аналогично оператору if в других языках программирования. Если истинно первое условие, то выполняется первый список команд, иначе — проверяется второе условие и т. д. Количество блоков elif, понятно, не ограничено.

Самая ответственная задача — это правильно составить условие. Условия записываются в квадратных скобках. Вот пример записи условий:

```
# Переменная N = 10
[ N==10 ]
# Переменная N не равна 10
[ N!=10 ]
```

Операции сравнения указываются не с помощью привычных знаков > или <, а с помощью следующих выражений:

- -lt — меньше;
- -gt — больше;
- -le — меньше или равно;
- -ge — больше или равно;
- -eq — равно (используется вместо ==).

Применять данные выражения нужно следующим образом:

```
[ переменная выражение значение|переменная ]
```

Например:

```
# N меньше 10
[ $N -lt 10 ]
# N меньше A
[ $N -lt $A ]
```

В квадратных скобках вы также можете задать выражения для проверки существования файла или каталога:

- `-e файл` — условие истинно, если файл существует;
- `-d каталог` — условие истинно, если каталог существует;
- `-x файл` — условие истинно, если файл является исполнимым.

С оператором `case` мы уже немного знакомы, но сейчас рассмотрим его синтаксис подробнее:

```
case переменная in
значение_1) команды_1 ;;
...
значение_N) команды_N ;;
*) команды_по_умолчанию;;
esac
```

Значение указанной переменной по очереди сравнивается с приведенными значениями (`значение_1` ... `значение_N`). Если есть совпадение, то будут выполнены команды, соответствующие значению. Если совпадений нет, то будут выполнены команды по умолчанию. Пример использования `case` был приведен в листинге 2.2.

2.9. Функции

В `bash` можно использовать функции. Синтаксис объявления функции следующий:

```
имя() { список; }
```

Вот пример объявления и использования функции:

```
list_txt()
{
echo "Выводим текстовые файлы"
ls *.txt
}
```

2.10. Примеры сценариев

2.10.1. Сценарий мониторинга журнала

Начнем с простого сценария мониторинга журнала (листинг 2.3). Системные журналы постоянно обновляются, а наш сценарий будет каждые три секунды выводить последние 15 строк выбранного нами журнала. Сценарий будет полезен при настройке системы, когда нужно постоянно просматривать журналы, чтобы понять, как система реагирует на новые настройки. Для прекращения работы сценария следует нажать клавиши <Ctrl>+<C>.

Листинг 2.3. Мониторинг системного журнала

```
#!/bin/bash
# Интервал обновления, в секундах
INT=3
```

```
while [ true ]
do
# Выводим последние 15 строк журнала
tail -n 15 $1
# Ждем
sleep $INT
# Две пустые строки
echo; echo
done
```

Формат вызова следующий:

```
./сценарий файл_журнала
```

Например:

```
./script /var/log/messages
```

2.10.2. Переименование файлов

Следующий сценарий сложнее: он ищет файлы в текущем каталоге, в именах которых есть пробелы, и заменяет пробелы на символы подчеркивания (листинг 2.4).

Листинг 2.4. Сценарий rename_blanks

```
#! /bin/bash
#

# Сколько файлов мы переименовали
num=0
# Перебираем все файлы в текущем каталоге
for filename in *
do
# Передаем имя файла фильтру grep
# Если имя файла содержит пробел, то код завершения
# последней операции равен 0
     echo "$filename" | grep -q " "
     if [ $? -eq 0 ]
     then
# Если код завершения равен 0, переименовываем
# файл
       fname=$filename
       n=`echo $fname | sed -e "s/ /_/g"`
       mv "$fname" "$n"
       let "num += 1"
     fi
done
```

```
echo "Переименовано файлов: $num"

exit 0
```

2.10.3. Преобразование систем счисления

Сейчас мы напишем простенький сценарий, преобразующий десятичное число в шестнадцатеричное с помощью программы dc (листинг 2.5). Самим преобразованием будет заниматься программа dc, а наш сценарий только подготовит данные для этой программы.

Листинг 2.5. Сценарий dec_hex

```
#!/ bin/bash
B=16   # Основание системы счисления
# Проверяем, является ли параметр $1 числом
if [ -z "$1" ]
then
echo "Использование: dec_hex число"
exit 1
fi
# Передаем число ($1), систему счисления
# программе dc
echo ""$1" "$B" o p" | dc
```

ГЛАВА 3

Создание сценариев на *tcsh*

3.1. Использование *tcsh*

Как уже было отмечено в *главе 1*, изначально оболочка `tcsh` была разработана для операционной системы TENEX, которая использовалась в далеком прошлом на компьютерах DEC PDP-10.

В некоторых дистрибутивах, например в Ubuntu, `tcsh` по умолчанию не установлена. Для ее установки нужно ввести команду:

```
sudo apt-get install tcsh
```

Сейчас не хочется делать ни экскурс в историю, ни проводить сравнение с `csh`. Скажу только, что во всех свободных системах (Linux, FreeBSD) файл /bin/csh — это ссылка на /bin/tcsh.

Как и `bash`, `tcsh` позволяет создавать сценарии, а язык `tcsh` насчитывает 65 встроенных команд. Просмотреть список этих команд можно командой `builtins`. На рис. 3.1 приведен вывод этой команды.

Основные интерактивные возможности `tcsh` следующие:

- редактирование командной строки;
- дополнение слов — как команд, так и путей;
- хранение и возможность просмотра истории команд;
- управление задачами.

Редактирование командной строки осуществляется с помощью клавиш <Left> и <Right> (перемещение курсора по командной строке) и клавиш <Delete> и <Backspase> (удаление символов). Как видите, все просто.

Для автоматического дополнения слов, как и в `bash`, используется клавиша <Tab>, но правильнее использовать комбинации клавиш <Ctrl>+<I> и <Ctrl>+<D>. Первая обеспечивает автодополнение слова, а вторая выводит список подходящих вариантов для автодополнения.

```
denis-desktop:~> builtins
:          @          alias      alloc      bg         bindkey    break
breaksw    builtins   case       cd         chdir      complete   continue
default    dirs       echo       echotc     else       end        endif
endsw      eval       exec       exit       fg         filetest   foreach
glob       goto       hashstat   history    hup        if         jobs
kill       limit      log        login      logout     ls-F       nice
nohup      notify     onintr     popd       printenv   pushd      rehash
repeat     sched      set        setenv     settc      setty      shift
source     stop       suspend    switch     telltc     termname   time
umask      unalias    uncomplete unhash     unlimit    unset      unsetenv
wait       where      which      while
denis-desktop:~>
```

Рис. 3.1. Список встроенных команд `tcsh`

Просмотреть историю ранее введенных команд можно с помощью клавиш <Up> и <Down>. Если же эти клавиши почему-то не работают (меньше нужно играть в NFS!), то можно использовать команду `history`:

```
$ history
1       13:08   clear
2       13:09   builtins
3       13:19   history
```

Вызвать любую команду из истории можно с помощью конструкции `!#`, где `#` — это номер команды, например:

```
!1
```

Управление задачами осуществляется так же, как и в `bash`. Чтобы запустить команду в фоновом режиме, нужно добавить к ней символ `&`, например:

```
$ command &
```

Вывести список запущенных в фоновом режиме задач можно командой `jobs`. Команда `fg` переводит задачу в активный режим (foreground), нужно указать ее номер (полученный командой `jobs`), например:

```
$ fg 1
```

Команда `bg` переводит активную задачу в фоновый режим (background). Как и для команды `fg`, нужно указать номер задачи:

```
$ bg 1
```

3.2. Конфигурационные файлы *tcsh*

Основные глобальные файлы конфигурации `tcsh` — /etc/csh.cshrc, /etc/csh.login, /etc/csh.logout. Первый файл считывается при запуске каждого экземпляра `tcsh` в интерактивном режиме, второй — только если `tcsh` является оболочкой по умолчанию для запустившего ее пользователя. Третий файл считывается при выходе из `tcsh` при условии, что `tcsh` является оболочкой по умолчанию для этого пользователя (является так называемым login shell).

При регистрации пользователя в домашнем каталоге создаются файлы ~/.cshrc, ~/.login и ~/.logout, которые являются пользовательскими аналогами /etc/csh.cshrc и /etc/csh.login. Содержимое этих файлов копируется из каталога /etc/skel (или /usr/share/skel/ — в некоторых системах).

Порядок считывания глобальных и пользовательских конфигурационных файлов задается при компиляции `tcsh` и может отличаться в разных системах. Обычно сначала считываются глобальные файлы, а затем пользовательские.

Также в пользовательском каталоге есть файл ~/.history, содержащий историю введенных команд.

В конфигурационных файлах устанавливаются псевдонимы команд (команда `alias`), переменные окружения (команда `setenv`), служебные переменные `tcsh` (команда `set`), влияющие на поведение оболочки. Определять переменные окружения имеет смысл в файле ~/.login, который считывается только один раз — при входе пользователя в систему, а файл ~/.cshrc перечитывается при запуске каждого экземпляра.

Рассмотрим пример определения псевдонимов команд:

```
alias hist history 25
alias rm rm -i
```

Первая команда создает псевдоним `hist`, выводящий последние 25 команд истории. Вторая команда создает псевдоним для команды `rm`, который называется так же — `rm`, а параметр `-i` означает, что при удалении файлов будет сделан запрос.

Вот пример использования команды `set`, которая устанавливает значение переменной `path` (путь поиска программ):

```
set path = (/bin /usr/bin /usr/local/bin /usr/X11R6/bin)
```

Переменные окружения устанавливаются командой `setenv` (в файле ~/.login), например:

```
setenv EDITOR nano
```

Здесь мы установили переменную окружения `EDITOR`, задающую текстовый редактор по умолчанию.

3.3. Создание сценариев на *tcsh*

3.3.1. Переменные, массивы и выражения

Переменные, как уже было показано, устанавливаются командой `set`:

```
set name=denis
```

Здесь мы установили переменную `name` (значение — `denis`). Вывести значение переменной можно так:

```
echo $name
```

```
denis-desktop:~> set
_       jobs

addsuffix
argv    ()
autoexpand
autolist
csubstnonl
cwd     /home/denix
dirstack        /home/denix
echo_style      both
edit
gid     1000
group   denix
history 100
home    /home/denix
killring        30
owd
path    (/usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin /bin /usr/games
)
prompt  %U%m%u:%B%~%b%#
prompt2 %R?
prompt3 CORRECT>%R (y|n|e|a)?
shell   /usr/bin/tcsh
```

Рис. 3.2. Команда `set`

Если вы введете просто команду `set` (без параметров), то вы увидите список уже установленных переменных, в том числе и служебных (рис. 3.2).

Удалить переменную можно командой `unset`:

```
unset name
```

Тогда при обращении к переменной вы увидите сообщение:

```
name: Undefined variable.
```

Переменные окружения принято устанавливать командой `setenv` так:

```
setenv переменная значение
```

Обратите внимание, что между именем переменной и значением нет знака равенства. Пример:

```
setenv  EDITOR  nano
```

В `tcsh` также можно создавать массивы. Вот пример создания и использования массива:

```
$ set nums = (one two three four five)

$ echo $nums
one two three four five

$ echo $nums[3]
three

$ echo $nums[1-3]
one two three
```

Как видите, мы можем вывести сразу весь массив, конкретный элемент и диапазон элементов, что очень удобно. Нумерация элементов массива начинается с 1.

Отдельного разговора заслуживают числовые переменные. Чтобы присвоить переменной число или результат арифметического выражения, нужно использовать символ `@`, при этом символ `$` перед именем переменной указывать не нужно.

Рассмотрим несколько примеров:

```
$ @ num = 0
$ echo $num
0
$ @ num = ( 2 + 2 ) * 2
$ echo $num
8
$ @ num += 5
$ echo $num
13
$ @ num++
$ echo $num
14
$ @ num2 = 5
$ @ num = $num2 + 5
$ echo $num
10
$ @ num = 1
echo $nums[$num]
one
```

Когда вы используете переменные в выражениях, тогда нужно указывать символ `$`, а после символа `@` указывать `$` не нужно. После `@` обязательно должен быть пробел.

Общий синтаксис при работе с числовыми переменными выглядит так:

```
@ переменная оператор выражение
```

Оператор — это один из операторов присваивания C: `=`, `+=`, — `=`, `*=`, `/=` или `%=`. Выражение — это арифметическое выражение, которое тоже строится по тем же правилам, что и в языке C. Ничего удивительного: оболочка `tcsh` предназначена для тех, кто знаком с языком C, и должна была облегчить создание сценариев C-программистам.

Теперь рассмотрим пример работы с массивами чисел:

```
$ set a = (0 0 0 0 0)
$ @ a[1] = 10
$ @ a[3] = ($ages[1] + 5)
$ echo $a[3]
15
```

3.3.2. Чтение ввода пользователя

Для чтения ввода пользователя используется конструкция "$<", например:

```
echo -n "Введите строку: "
set line = "$<"
```

3.3.3. Переменные оболочки *tcsh*

При создании сценариев на tcsh вы можете использовать переменные оболочки, представленные в табл. 3.1.

Таблица 3.1. *Переменные оболочки* tcsh

Переменная	Описание
argv	Массив содержит аргументы командной строки (параметры, переданные сценарию). argv[1] — первый параметр, а argv[0] — это имя самого сценария. Обратиться к параметрам можно через конструкцию argv[n] или $n, например argv[1] или $1. Элемент массива argv[*] содержит все параметры вместе
autolist	Контролирует завершение команд и переменных. См. man tcsh
autologout	Включает возможность автоматического выхода. По умолчанию — 60 минут, если вы суперпользователь. Данная переменная не устанавливается (по умолчанию) для обычных пользователей
cdpath	Влияет на команду cd, задает путь поиска команд, обычно устанавливается в файле ~/.login, например: set cdpath = (/home/denix /home/denix/bin)
cwd	Содержит имя текущего каталога
fignore	Содержит массив суффиксов, которые tcsh будет игнорировать при завершении имен файлов
gid	Содержит идентификатор группы
histfile	Переменная содержит полное имя файла, в котором находится история команд. По умолчанию ~/.history
history	Размер списка истории
home или HOME	Имя домашнего каталога пользователя
mail	Содержит имя файла и каталога для проверки почты. TC Shell каждые 10 минут проверяет почту
owd	Предыдущий рабочий каталог
path или PATH	Путь поиска программ, обычно устанавливается в конфигурационных файлах tcsh: set path = (/usr/bin /bin /usr/local/bin /usr/bin/X11 ~/bin .)
prompt	Содержит формат приглашения командной строки, аналогична переменной PS1 в bash. Модификаторы формата описаны в табл. 3.2. Значение по умолчанию: set prompt = '! $ '

Таблица 3.1 *(окончание)*

Переменная	Описание
prompt2	Содержит формат приглашения командной строки для управляющих структур while и foreach
prompt3	Содержит формат приглашения командной строки при проверке правописания
savehist	Количество команд, которые будут сохранены в файл истории
shell	Полный путь к исполнимому файлу оболочки
status	Код завершения последней выполненной команды
tcsh	Версия tcsh
time	Может содержать только число или же буквы и цифры. Если переменная содержит только число, то это количество секунд процессорного времени. Если выполнение какой-то команды заняло больше времени, чем указано в time, то после выполнения команды будет выведено, сколько времени она занимала. Если time = 0, то после каждой команды будет выведено время выполнения. Если time содержит буквы и цифры (табл. 3.3), то это просто формат времени
user	Имя пользователя

Таблица 3.2. *Формат командной строки*

Модификатор	Описание
%/	Текущий каталог
%~	То же, что и %/, но заменяет путь к домашнему каталогу пользователя тильдой
%! или %h или !	Текущий номер события
%m	Имя узла без домена
%M	Полное имя узла, с доменом
%n	Имя пользователя
%t	Время дня
%p	Время дня (с секундами)
%d	День недели
%D	День месяца
%W	Месяц, формат мм
%y	Год, формат гг
%Y	Год, формат гггг
%#	Решетка (#) для суперпользователя или знак больше (>) для обычного пользователя
%?	Код завершения последней команды

Таблица 3.3. *Формат времени*

Модификатор	Описание
%U	Время, проведенное командой в пользовательком режиме (в процессорных секундах)
%S	Время, проведенное командой в режиме ядра (в процессорных секундах)
%W	Сколько раз процесс команды был выгружен на диск
%X	Средний размер сегмента кода программы, в килобайтах
%D	Средний объем памяти, используемый командой, в килобайтах
%K	Общий размер памяти, занятый командой (считается как %X+%D), в килобайтах
%M	Максимальный объем памяти, занятый командой, в килобайтах
%F	Количество ошибок страниц памяти
%I	Количество операций ввода
%O	Количество операций вывода

Перед тем как приступить к изучению управляющих структур, рассмотрим применение скобок в tcsh. Предположим, есть переменная:

```
$ set aa=abra
```

Потом нам нужно вывести ее в составе строки, для этого мы будем использовать фигурные скобки:

```
$ echo ${aa}cadabra
abracadabra
```

3.3.4. Управляющие структуры

Условный оператор *if*

Синтаксис оператора if очень прост:

```
if (выражение) команда
```

Команда будет выполнена, если выражение истинно. Выражения формируются так же, как в языке C. В листинге 3.1 представлен небольшой сценарий, проверяющий количество аргументов, переданных ему.

Листинг 3.1. Первый сценарий на tcsh

```
#!/bin/tcsh
if ( $#argv == 0 ) echo "Аргументы не заданы"
```

Также можно использовать следующее выражение:

```
-n имя_файла
```

В данном случае возможные варианты `n` представлены в табл. 3.4.

Таблица 3.4. *Значения* `n`

n	Описание
b	Файл является блочным устройством (обмен данными с устройством осуществляется блоками данных)
c	Файл является символьным устройством (обмен данными с устройством осуществляется посимвольно)
d	Файл является каталогом
e	Файл существует
g	Для файла установлен бит SGID
k	Для файла установлен "липкий" бит
l	Файл является символической ссылкой
o	Файл принадлежит текущему пользователю
p	Файл является именованным потоком (FIFO)
r	У пользователя есть право чтения файла
s	Файл не пустой (ненулевой размер)
S	Файл является сокетом
t	Дескриптор файла открыт и подключен к экрану
u	Для файла установлен SUID
w	У пользователя есть право записи файла
x	У пользователя есть право выполнения файла
X	Файл является встроенной командой или его исполнимый файл найден при поиске в каталогах, указанных в `$path`
z	Файл пуст (нулевой размер)

Рассмотрим пример использования данного условия:

```
if -e $1 echo "Файл существует"
```

Условный оператор *if..then..else*

Условный оператор `if..then..else` похож на `if`, только добавляется блок `else` (*иначе*), который выполняет команды в случае ложности условия. Сокращенная версия оператора выглядит так:

```
if (выражение) then
    команды, которые будут выполнены в случае истинности выражения
endif
```

Полная версия оператора выглядит так:

```
if (выражение) then
    команды, которые будут выполнены в случае истинности выражения
else
    команды, которые будут выполнены в противном случае (когда выражение = false)
endif
```

Существует еще одна форма этого оператора, точнее, это отдельный оператор `if..then..elif`:

```
if (выражение1) then
     команды (если выражение1 = true)
else if (выражение2) then
    команды (если выражение2 = true)
. . .
else
     команды (если ни одно из выражений не равно true)
endif
```

Рассмотрим небольшой пример использования условного оператора (листинг 3.2).

Листинг 3.2. Пример использования условного оператора

```
#!/bin/tcsh
# Получаем число из командной строки
set num = $argv[1]
set flag
#
if ($num < 0) then
    @ flag = 1
else if (0 <= $num && $num < 50) then
    @ flag = 2
else if (50 <= $num && $num < 1000) then
    @ flag = 3
else
    @ flag = 4
endif
#
echo "Flag: ${flag}."
```

Оператор *foreach*

Оператор `foreach` удобно использовать для перебора массивов, его синтаксис:

```
foreach индекс-цикла (список-аргументов)
     команды
end
```

Также цикл `foreach` удобно использовать при работе с файлами, например:

```
foreach f ( *.txt )
   echo $f
end
```

Здесь мы в цикле выводим имена всех текстовых файлов в текущем каталоге. Конечно, для этой цели проще вызвать команду `ls`, но здесь мы продемонстрировали использование `foreach`.

Оператор *while*

Оператор `while` — это еще один вариант цикла. Команды, находящиеся в теле цикла, выполняются до тех пор, пока выражение истинно:

```
while (выражение)
    команды
end
```

Рассмотрим пример сценария, вычисляющего факториал числа `n`, при этом `n` задается в командной строке (листинг 3.3).

Листинг 3.3. Пример использования оператора `while`

```
#!/bin/tcsh
set n = $argv[1]
set i = 1
set fact = 1
#
while ($i <= $n)
    @ fact *= $i
    @ i++
end
#
echo "Факториал числа $n равен $fact"
```

В циклах вы можете использовать операторы `break` и `continue`. Оператор `break` прерывает цикл и передает управление оператору, следующему за `end`. Оператор `continue` прерывает только текущую итерацию цикла и передает управление оператору `end`, который после получения управления начнет следующую итерацию цикла.

Оператор *switch*

В ряде случаев использовать оператор `switch` намного удобнее, чем серию условных операторов. Синтаксис `switch` следующий:

```
switch (строка)
    case образец1:
        команды1
    breaksw
```

```
    case образец2:
        команды2
    breaksw
    ...
    default:
        команды по умолчанию
    breaksw
endsw
```

Вот пример использования данного оператора:

```
set string = test
switch (string)
       case test:
              echo "Строка: test"
       breaksw
       case text:
              echo "Строка: text"
       breaksw
       default:
              echo "Строка не опознана"
       breaksw
endsw
```

3.3.5. Встроенные команды *tcsh*

Оболочка `tcsh` обладает встроенными командами (как, впрочем, и любая другая оболочка). В табл. 3.5 приведены самые полезные встроенные команды оболочки `tcsh`.

Таблица 3.5. *Самые полезные встроенные команды* `tcsh`

Команда	Описание
`@`	Вычисляет арифметическое выражение. Данная команда была подробно описана ранее при написании сценариев
`alias`	Создает псевдоним для команд оболочки, эту команду мы тоже рассмотрели ранее
`alloc`	Отображает отчет о количестве свободной и использованной памяти
`bg`	Перемещает приостановленную задачу в фоновый режим
`builtins`	Отображает список встроенных команд
`cd` или `chdir`	Изменяет текущий каталог
`dirs`	Отображает стек каталогов
`echo`	Отображает свои аргументы. Обычно используется при написании сценариев
`exec`	Запускает другую программу в этой же оболочке

Таблица 3.5 *(продолжение)*

Команда	Описание
`exit`	Осуществляет выход из `tcsh`
`fg`	Перемещает задачу на "передний план", т. е. делает ее активной
`filetest`	В основном используется при написании сценариев. Позволяет проверить тип файла (устройство, каталог, файл), существование файла и осуществить ряд других полезных проверок (см. табл. 3.4)
`glob`	Похожа на `echo`, но не отображает пробелы между аргументами и не выводит символ новой строки после своего вывода
`hashstat`	Выводит статистику механизма хэширования `tcsh`
`history`	Отображает введенные ранее команды
`jobs`	Отображает список задач (приостановленные команды и те, которые выполняются в фоновом режиме)
`kill`	Завершает задачу или процесс
`limit`	Позволяет ограничить ресурсы текущего процесса и всех процессов, которые он будет создавать
`login`	Используется для входа пользователя. Сопровождается именем пользователя
`logout`	Завершает сессию, если вы используете `tcsh` в качестве оболочки по умолчанию
`ls-F`	Похожа на `ls -F`, но выполняется быстрее
`nice`	Позволяет изменить приоритет процесса
`nohup`	Позволяет вам выйти из системы без завершения процессов, выполняемых в фоновом режиме
`notify`	Уведомляет вас при изменении статуса одной из ваших задач
`popd`	Удаляет каталог из стека каталогов
`printenv`	Отображает список всех переменных окружения
`pushd`	Изменяет рабочий каталог и помещает новый каталог на вершину стека каталогов
`rehash`	Пересоздает внутренние таблицы, используемые механизмом хэширования. Допустим, вы создали свой сценарий и поместили его в /usr/bin. Оболочка `tcsh` не увидит этот файл до тех пор, пока вы не обновите внутренние таблицы
`repeat`	Команде нужно передать два аргумента — количество повторений и простую команду (без потоков и списка команд) и повторяет эту команду указанное количество раз
`sched`	Выполняет команду в указанное время, например: `$ sched 10:00 echo "Уже 10 часов!"`
`set`	Объявляет и инициализирует локальную переменную
`setenv`	Объявляет и инициализирует переменную окружения
`source`	Запускает сценарий оболочки, указанный в качестве аргумента. Данная команда не порождает новый процесс, а просто обрабатывает сценарий. Данную команду можно использовать как `include` в обычных языках программирования

Таблица 3.5 *(окончание)*

Команда	Описание
`stop`	Останавливает задачу или процесс (которая или который выполняется в фоновом режиме)
`suspend`	Приостанавливает текущую оболочку и помещает ее в фоновый режим
`time`	Запускает команду, указанную в качестве параметра. После выполнения команды отображает информацию о ее выполнении (время выполнения)
`umask`	Изменяет маску прав доступа по умолчанию (см. `man umask`)
`unalias`	Удаляет псевдоним команды
`unhash`	Выключает механизм хэширования. См. также `hashstat` и `rehash`
`unlimit`	Удаляет лимиты текущего процесса
`unset`	Удаляет объявление переменной
`unsetenv`	Удаляет объявление переменной окружения
`where`	В качестве аргумента нужно передать команду, после чего получите информацию о том, является ли команда встроенной или исполнимым файлом и где находится этот файл. Похожа на команду `which`
`which`	Подобна стандартной утилите `which`, но работает быстрее и владеет информацией обо всех командах и псевдонимах. Выводит местонахождение исполнимого файла, если он вообще существует. Поиск производится в каталогах, указанных в переменной окружения `path`

ГЛАВА 4

Пакет dialog: псевдографический интерфейс пользователя

4.1. Необходимость в графическом интерфейсе

В двух предыдущих главах мы рассмотрели встроенные языки двух командных оболочек — `bash` и `tcsh`. Все бы хорошо, но пользователю хочется предоставить графический интерфейс. Одно дело, когда создаешь сценарий командной оболочки для себя — можно мириться даже с отсутствием какого-либо интерфейса пользователя, а все параметры либо передавать в командной строке, либо читать из файла конфигурации. Но обычные пользователи панически боятся черного, как в DOS, экрана — все сообщения, дабы не травмировать психику пользователя, должны выводиться в графических окнах, варианты ответа Да/Нет должны выбираться либо клавишами управления курсором, либо мышью, обязательно должна быть графическая шкала процесса выполнения и т. д.

Но где в командной строке взять графику? Ведь это уже другой уровень — неужели придется разрабатывать приложение для X.Org? Выход есть — это пакет dialog, позволяющий реализовать простейший псевдографический интерфейс пользователя. Однако не нужно возлагать на dialog больших надежд — его возможности довольно невелики, однако диалог с пользователем с его помощью организовать можно, а от него большего и не требуется. Другими словами, интерфейс в стиле Midnight Commander средствами dialog не создашь, однако псевдографические окна (создаются с помощью символов псевдографики без перехода в графический режим видеокарты) dialog создавать позволяет.

Пакет dialog по умолчанию не устанавливается, однако входит в состав большинства дистрибутивов Linux. Для его установки воспользуйтесь менеджером пакетов вашего дистрибутива. Например, для установки dialog в OpenSUSE введите команду:

```
sudo zypper install dialog
```

В Ubuntu используется `apt-get`:

```
sudo apt-get install dialog
```

Далее будут рассмотрены типичные примеры применения пакета dialog в сценариях.

4.2. Простейшее диалоговое окно

Начнем с простейшего диалогового окна, обычно отображающего какой-нибудь вопрос и предоставляющего пользователю право выбора — кнопки **Да** и **Нет**. Сначала рассмотрим код сценария (листинг 4.1), а потом будем разбираться, что и к чему.

Листинг 4.1. Сценарий yes-no

```
#!/bin/bash
DIALOG=${DIALOG=dialog}

$DIALOG --title " Самый простой диалог " --clear \
        --yesno "Обычное диалоговое окно с кнопками Да/Нет" 10 40

case $? in
    0)
        echo "Пользователь выбрал 'Да'.";;
    1)
        echo "Пользователь выбрал 'Нет'.";;
    255)
        echo "Пользователь нажал ESC.";;
esac
```

Первые две строки будут постоянными для сценариев, использующих dialog. Далее (в четвертой строке) вызывается программа `DIALOG` с параметрами `--title`, `--clear`, `--yesno`. Параметр `--title` задает заголовок окна, параметр `--clear` очищает экран перед выводом диалога, параметр `--yesno` отображает диалоговое окно с кнопками **Да**/**Нет**. После параметра `--yesno` следует строка, которая будет отображена в диалоговом окне, а числа 10 и 40 задают высоту и ширину диалога в знакоместах.

В большинстве случаев программа `DIALOG` вызывается именно так, как показано в листинге 4.1: указывается заголовок и очищается экран. Разница лишь в типе диалога и параметрах диалога, которые следуют после параметра, задающего тип диалога (в нашем случае — `--yesno`).

Когда пользователь сделает выбор, программа `DIALOG` завершит свою работу и возвратит в сценарий значение. Положительному ответу пользователя (**Да** или **ОК** в диалоговом окне) соответствует значение 0, отрицательному — 1. Если пользователь нажал клавишу <Esc>, в сценарий возвращается значение 255.

Наш несложный сценарий просто выводит сообщение в зависимости от выбора пользователя. Вы же вместо `echo` можете произвести определенные действия. Напомню, что оператор `case` подробно рассматривался в *главе 2*.

Сделаем файл yes-no исполнимым и запустим его:

```
chmod +x yes-no
./yes-no
```

Результат работы сценария изображен на рис. 4.1.

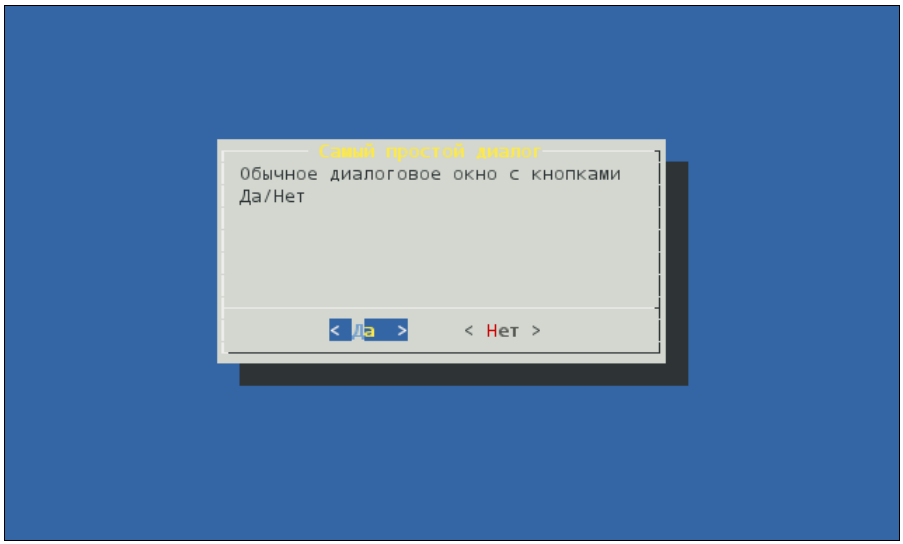


Рис. 4.1. Результат работы сценария yes-no

4.3. Информационное окно

Теперь напишем сценарий, выводящий обычное информационное окно. В этом окне будет отображен текст для пользователя и кнопка **OK**. Задача такого окна — обратить внимание пользователя, например "Нажмите OK для перезагрузки компьютера" — перед перезагрузкой желательно отобразить такое сообщение, чтобы она не произошла внезапно. В листинге 4.2 приведен простейший вариант информационного окна. Обратите внимание: в информационном сообщении используется последовательность `\n` для начала новой строки.

Листинг 4.2. Сценарий msg

```
#!/bin/bash
DIALOG=${DIALOG=dialog}

$DIALOG --title "Сообщение" --msgbox "Обычное информационное окно,\nждет, пока
пользователь нажмет Enter" 10 40

case $? in
0)
       echo "Пользователь нажал OK";;
255)
       echo "Нажата Esc";;
esac
```

Результат работы нашего сценария приведен на рис. 4.2. Но у этого сценария есть один недостаток. Представим, что вы разрабатываете простейший инсталлятор для

вашего дистрибутива Linux. Проблема в том, что если пользователь на время установки отойдет куда-то, например выпить кофе, то компьютер не перезагрузится до тех пор, пока кто-нибудь не нажмет <Enter>. Нужно сделать небольшой таймер. Это выходит, конечно, за рамки использования пакета dialog, но на практике пригодится многим программистам.

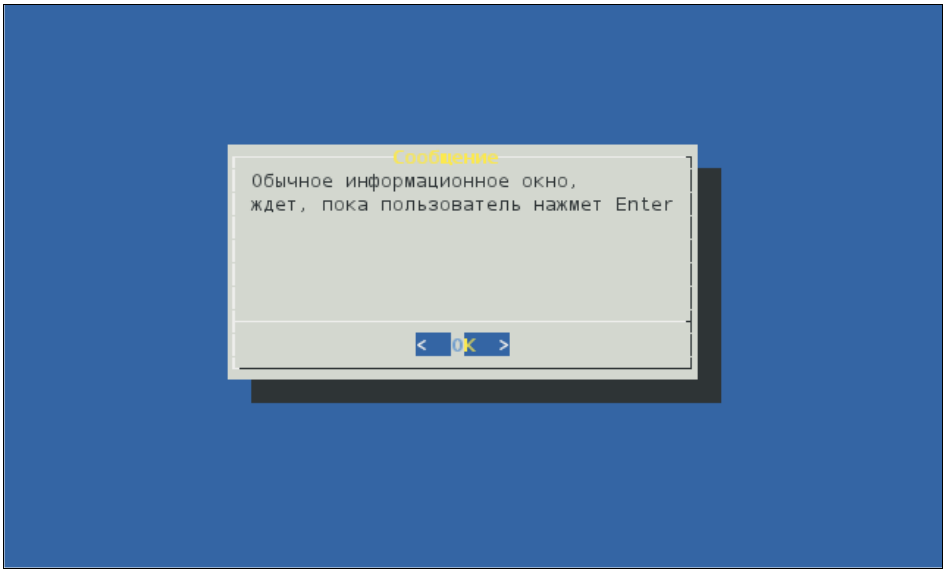


Рис. 4.2. Результат работы сценария msg

Сейчас мы отобразим информационное окно с таймером обратного отсчета до перезагрузки. Для нашей цели, правда, мы будем использовать не `msgbox`, а `infobox` — он больше подходит. Разница — в отсутствии каких-либо кнопок: все равно, раз выбор сделать нельзя, то и кнопки не нужны. Сценарий info представлен в листинге 4.3, а результат его работы — на рис. 4.3.

Листинг 4.3. Сценарий info

```
#! /bin/bash
DIALOG=${DIALOG=dialog}

# К-во секунд до перезагрузки
left=10
# Единица времени
unit="секунд"

while test $left != 0
do
$DIALOG --sleep 1 \
--title "Установка завершена" \
```

```
--infobox "Установка системы успешно завершена!\nДо перезагрузки осталось $left
$unit " 10 40
left=`expr $left – 1`
test $left = 1 && unit="секунд"
done
# Перезагрузка (раскомментируйте, если нужно):
# /sbin/reboot
```

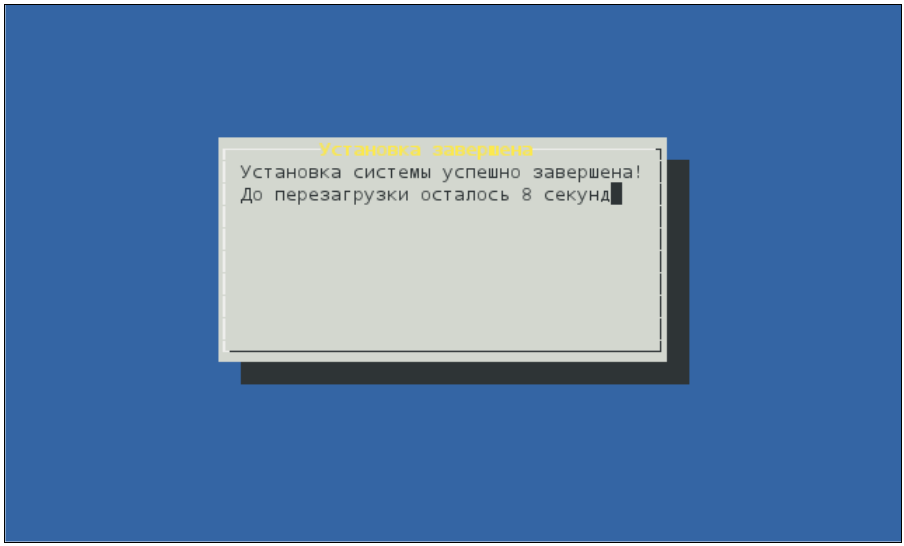


Рис. 4.3. Результат работы сценария info

Если вы внимательно прочитали листинг 4.3, а еще лучше — попробовали на практике, то у вас возник вопрос: что делает параметр `sleep`? Пока вы читали предыдущее предложение, уже, думаю, догадались. Параметр `sleep` заставляет dialog перерисовывать окно каждые *N* секунд, в нашем случае *N* = 1.

4.4. Ввод текста

Возможности командного интерпретатора для ввода текста очень скудны — программа `read`, конечно, позволяет ввести текстовую строку, но ваш сценарий будет выглядеть гораздо эффектнее, если у вас будет красивое текстовое поле. Сценарий input, создающий поле для ввода текста (рис. 4.4), представлен в листинге 4.4.

Листинг 4.4. Сценарий input

```
#!/bin/bash
DIALOG=${DIALOG=dialog}

tempfile=`tempfile 2>/dev/null` || tempfile=/tmp/test$$
trap "rm -f $tempfile" 0 1 2 5 15
```

```
$DIALOG --title "Ввод данных" --clear \
        --inputbox "Как вас зовут?" 16 51 2> $tempfile

retval=$?

case $retval in
  0)
    echo "Пользователь ввел: `cat $tempfile`"
    ;;
  1)
    echo "Пользователь отказался от ввода";;
  255)
    if test -s $tempfile ; then
      cat $tempfile
    else
      echo "Пользователь нажал ESC."
    fi
    ;;
esac
```

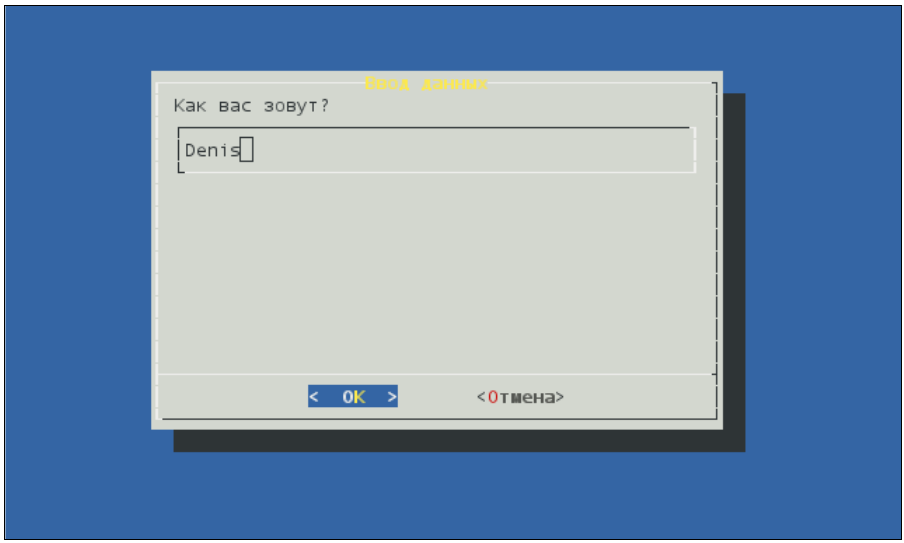


Рис. 4.4. Ввод данных (сценарий input)

После привычных двух строк следуют две строки, создающие временный файл, в который и будет записан ввод пользователя. Затем мы прочитаем этот файл, чтобы посмотреть, что ввел пользователь. По окончании работы сценария временный файл будет удален.

4.5. Создание меню

Все предложенные ранее диалоги хорошо использовать уже в процессе выполнения сценария, но в самом его начале неплохо бы отобразить меню, чтобы пользователь мог выбрать одну из команд. В листинге 4.5 приведен сценарий menu, реализующий простейшее меню, без которого не обходился ни один конфигуратор системы в ранних версиях дистрибутивов (рис. 4.5). Если вы когда-то работали с `linuxconf` и другими подобными конфигураторами в старых версиях Red Hat, Mandrake, то наверняка узнали привычное меню.

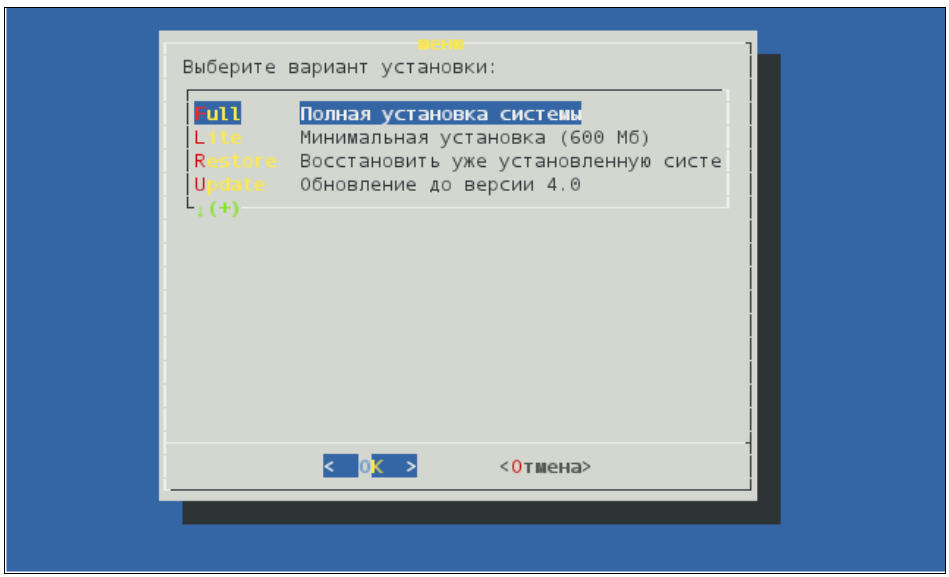


Рис. 4.5. Сценарий menu

Листинг 4.5. Сценарий menu

```
#!/bin/bash
DIALOG=${DIALOG=dialog}
tempfile=`mktemp 2>/dev/null` || tempfile=/tmp/test$$
trap "rm -f $tempfile" 0 1 2 5 15

$DIALOG --clear --title "Меню" \
        --menu "Выберите вариант установки:" 20 51 4 \
        "Full"  "Полная установка системы" \
        "Lite" "Минимальная установка (600 Мб)" \
        "Restore" "Восстановить уже установленную систему" \
        "Update" "Обновление до версии 4.0" \
        "Reboot"  "Перезагрузка" 2> $tempfile

retval=$?
```

```
choice=`cat $tempfile`

case $retval in
  0)
    echo "пользователь выбрал команду '$choice'";;
  1)
    echo "Пользователь отказался от ввода";;
  255)
    echo "Пользователь нажал ESC.";;
esac
```

Действие этого сценария основано на предыдущем сценарии ввода текста, только пользователь не вводит текст вручную, а выбирает один из предложенных ему вариантов из меню. Потом наш сценарий выводит команду меню, которую выбрал пользователь.

4.6. Проблема выбора: зависимые и независимые переключатели

Раз уж мы затронули задачу написания сценария установки системы, то наверняка нам понадобятся зависимые и независимые переключатели. Зависимые переключатели используются для выбора одного из вариантов, например: будем вызывать fdisk или уничтожим все данные на жестком диске. Независимые переключатели удобно использовать для выбора компонентов системы.

Рассмотрим простой сценарий, позволяющий выбрать тип разметки диска — вручную или автоматический (листинг 4.6, рис. 4.6).

Листинг 4.6. Сценарий radio

```
#! /bin/bash
DIALOG=${DIALOG=dialog}
tempfile=`mktemp 2>/dev/null` || tempfile=/tmp/test$$
trap "rm -f $tempfile" 0 1 2 5 15

$DIALOG --backtitle "СДЕЛАЙТЕ РЕЗЕРВНУЮ КОПИЮ ВСЕХ ВАЖНЫХ ДАННЫХ!" \
        --title "Разметка диска" --clear \
        --radiolist "Тип разметки диска" 20 61 5 \
        "Manual"  "Ручная разметка программой fdisk" off \
        "Auto"    "Все разделы жесткого диска будут удалены, будет создан один
раздел для системы" ON 2> $tempfile

retval=$?

choice=`cat $tempfile`
case $retval in
  0)
```

```
    echo "Выбор пользователя '$choice'";;
  1)
    echo "Пользователь отказался от ввода";;
  255)
    echo "Пользователь нажал ESC.";;
esac
```

Теперь разберемся, что и для чего используется в сценарии. Параметр `--backtitle`, не описанный ранее, используется для вывода текста сзади от диалогового окна — так называемый фоновый заголовок. Тип диалога — `radiolist` (зависимый переключатель). Для создания независимых переключателей (флажков) просто замените `radiolist` на `checklist`:

```
$DIALOG --title "Разметка диска" --clear \
        --checklist "Выбор компонентов для установки" 20 61 5 \
```

Каждый вариант списка переключателей описывается так же, как и вариант меню, но после строки с описанием варианта следует либо `on` (переключатель включен), либо `off` (переключатель выключен). Напомню, что в случае с независимыми переключателями могут быть активны (включены) несколько переключателей, а если вы создаете список зависимых переключателей, то активным может быть только один. Для выбора переключателя нужно выделить его с помощью клавиш управления курсора и нажать <Пробел>.

В файл выбранные значения записываются через пробел, каждый вариант — в кавычках. Например, если переделать сценарий в листинге 4.6 (использовать `checklist` вместо `radiolist`) и выбрать оба варианта, то в файл будет записана строка:

```
"Manual" "Auto"
```

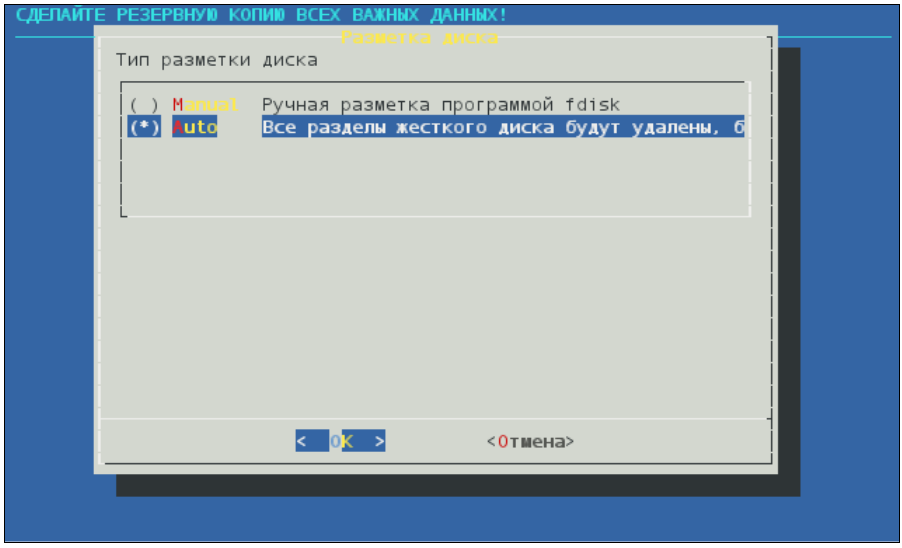

СДЕЛАЙТЕ РЕЗЕРВНУЮ КОПИЮ ВСЕХ ВАЖНЫХ ДАННЫХ!
Разметка диска
Тип разметки диска
( ) Manual Ручная разметка программой fdisk
(*) Auto Все разделы жесткого диска будут удалены, б
< OK >
<Отмена>

Рис. 4.6. Сценарий radio

4.7. Выбор даты и времени

Чего только не умудряются ввести пользователи, поэтому ввод пользователя нужно ограничить везде, где только можно. Например, для ввода даты можно было бы отобразить поле для ввода текста с подсказкой "Введите дату в формате ДД/ММ/ГГГГ", но, если честно, лично мне лень потом заниматься обработкой ввода пользователя с целью определить, а корректную ли он ввел дату.

Программа dialog существенно облегчает труд программиста, предоставляя средства для выбора даты и времени (рис. 4.7 и 4.8). В листинге 4.7 приведен сценарий calendar, сначала отображающий календарь для ввода даты, затем очищающий экран и отображающий окно выбора времени.

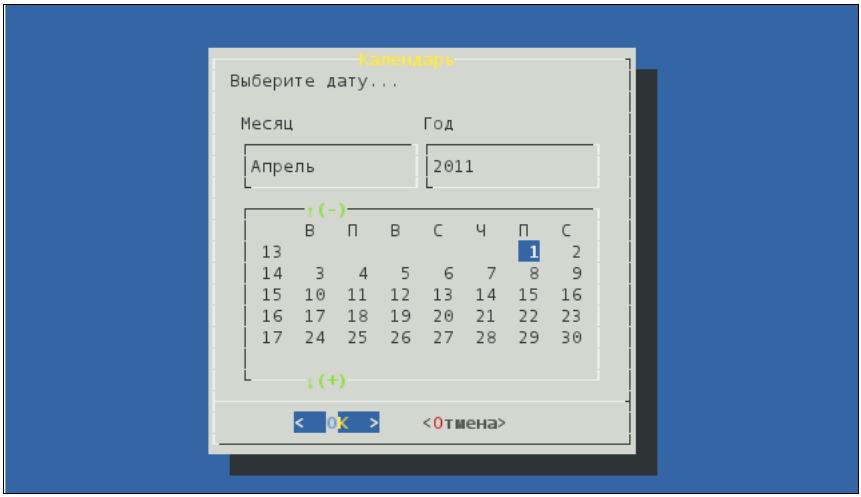


Рис. 4.7. Выбор даты (календарь)

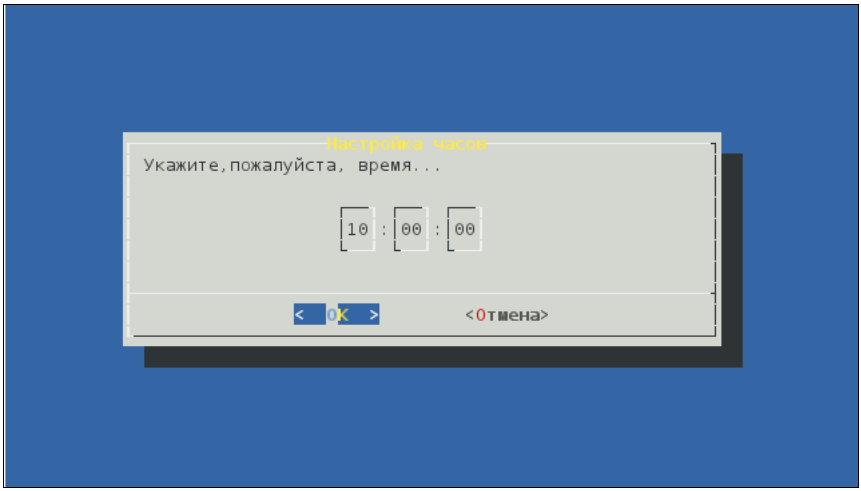


Рис. 4.8. Выбор времени

Листинг 4.7. Сценарий calendar

```
#!/bin/bash
DIALOG=${DIALOG=dialog}

USERDATE=`$DIALOG --stdout --title "Календарь" --calendar "Выберите дату..." 0
0 1 4 2011`

case $? in
  0)
    echo "Выбрана дата: $USERDATE.";;
  1)
    echo "Нажата кнопка Отмена.";;
  255)
    echo "Нажата клавиша ESC.";;
esac

clear

USERTIME=`$DIALOG --stdout --title "Настройка часов" \
           --timebox "Укажите,пожалуйста, время..." 0 0 10 00 00`

case $? in
  0)
    echo "Указано время: $USERTIME.";;
  1)
    echo "Нажата кнопка Отмена.";;
  255)
    echo "Нажата клавиша ESC.";;
esac
```

4.8. Индикатор

Для информирования пользователя о ходе процесса, например копирования файла или его обработки, целесообразно использовать индикатор (gauge). У нас никакого процесса не будет — мы просто заставим наш индикатор "двигаться": каждую секунду наша шкала будет "продвигаться" на 10%. На рис. 4.9 представлен сценарий gauge в действии.

Листинг 4.8. Сценарий gauge

```
#!/bin/bash
DIALOG=${DIALOG=dialog}

PCT=10
(
while test $PCT != 110
```

```
do
echo "XXX"
echo $PCT
echo "Выполнено: \n\
($PCT %)"
echo "XXX"
PCT=`expr $PCT + 10`
# засыпаем на 1 секунду, 1 секунда — это 10%
sleep 1
done
) | $DIALOG --title "Шкала" --gauge "Шкала" 20 70 0
```

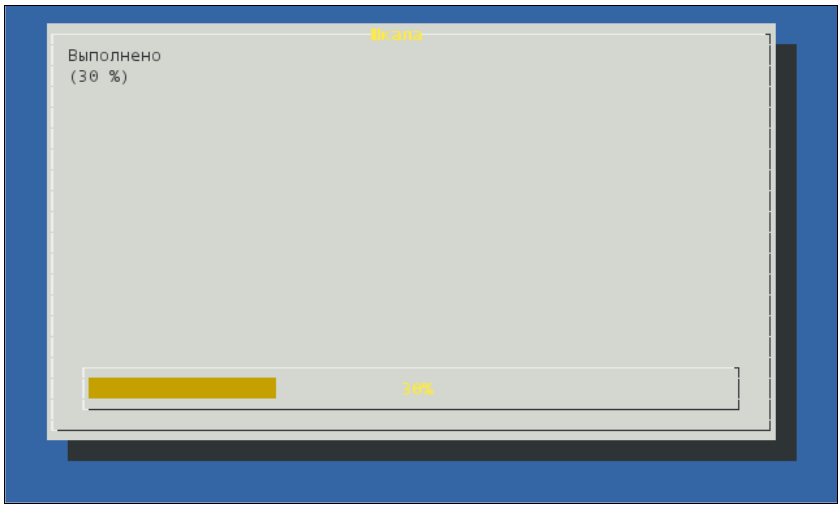


Рис. 4.9. Индикатор (сценарий gauge)

4.9. Диалог выбора файла

К вашим услугам еще один очень интересный диалог — диалог выбора файла (рис. 4.10). Диалог довольно удобный, но в нем есть одно "но". Если "стрелками" (клавишами управления курсором) выбрать файл и нажать <Enter>, то будет "выбран" только текущий каталог. Для выбора файла нужно нажать <Пробел>, чтобы имя файла появилось в поле внизу, а затем нажать <Enter>. Такая небольшая особенность может сбить с толку пользователя, особенно начинающего. В листинге 4.9 представлен сценарий, позволяющий выбрать файл.

Листинг 4.9. Сценарий fselect

```
#!/bin/bash
DIALOG=${DIALOG=dialog}

FILE=`$DIALOG --stdout --title "Выберите файл" --fselect $HOME/ 10 60`
```

```
case $? in
    0)
        echo "Выбран файл: \"$FILE\"";;
    1)
        echo "Пользователь нажал Отмена.";;
    255)
        echo "Пользователь нажал ESC.";;
esac
```

Рис. 4.10. Диалог выбора файла (сценарий fselect)

4.10. Дополнительные возможности

Существует близкий родственник пакета dialog — пакет Xdialog, позволяющий создавать элементы самого настоящего графического интерфейса. К преимуществам этого решения можно отнести запуск ваших `bash`-сценариев просто в графическом режиме, а не в терминале. А вот недостатков больше:

- нет той гибкости, которая обеспечивается другими скриптовыми языками, например парой TCL/Tk, которая также будет рассмотрена в этой книге (вы можете создавать только определенные типы по сути уже готовых диалоговых окон, но не можете сделать ничего, что выходило бы за рамки возможностей Xdialog);
- пакет Xdialog не всегда входит в состав дистрибутива (не говоря о том, что не устанавливается по умолчанию), поэтому могут возникнуть проблемы с переносом ваших сценариев. Если обычный dialog есть даже в самых современных дистрибутивах, то Xdialog придется искать в Интернете (желательно найти пакет именно для вашего дистрибутива) и устанавливать вручную. Например, для openSUSE пакет Xdialog доступен по ссылке: **http://www.novell.com/products/linuxpackages/opensuse/xdialog.html**;

- при переносе сценариев нужно будет выполнить некоторую настройку диалоговых окон (изменять их размеры), поскольку в пакете dialog они указывались в знакоместах, а не в пикселах.

Все созданные нами сценарии будут работать и при использовании Xdialog, но нужно заменить вторую строку каждого сценария — для Xdialog она выглядит так:

```
DIALOG=${DIALOG=Xdialog}
```

Если желаете написать универсальный сценарий, который при запуске в графическом режиме использовал бы Xdialog, а в консоли — обычный dialog, вместо второй строки используйте следующий код:

```
if [ -z $DISPLAY ]
then
    DIALOG=dialog
else
    DIALOG=Xdialog
fi
```

Конечно, еще нужно учитывать размеры диалогов. Самый простой способ — вообще их не указывать в надежде, что dialog/Xdialog установит их автоматически.

Пакет Xdialog содержит дополнительные типы диалогов (например, текстовый редактор exit-box). О дополнительных возможностях Xdialog вы сможете прочитать в документации, поставляемой с этим пакетом. Все рассмотренные в этой главе примеры доступны по адресу:

http://dkws.org.ua/mybooks/prg/dialog.tar.gz

ЧАСТЬ II

Основы программирования на C в Linux

Глава 5.	Компилятор *gcc* и вспомогательные программы
Глава 6.	Библиотеки. Автоматическая сборка библиотек
Глава 7.	Переменные окружения
Глава 8.	Ввод/вывод в Linux

Как уже было отмечено во *введении*, подразумевается, что читатель знаком с синтаксисом языка C, поскольку в этой книге основы C рассматриваться не будут, зато будет приведено все (или почти все), что касается разработки приложений на C в Linux. Во второй части книги мы поговорим о компиляторе `gcc`, библиотеках, переменных окружения и рассмотрим основные операции с файлами.

ГЛАВА 5

Компилятор *gcc* и вспомогательные программы

5.1. Выбор редактора

Прежде чем мы приступим к написанию программы и ее компиляции, нужно выбрать текстовый редактор. Этот вопрос в Linux заслуживает особого внимания, и сейчас вы поймете почему.

В любой UNIX-системе, в том числе и Linux, вы найдете стандартный текстовый редактор `vi`, но я не уверен, что вы будете его использовать. Начинающего программиста, который только-только "перекочевал" в Linux, эта программа в состоянии повергнуть в шок или, по крайней мере, ввести в ступор. Попробуйте запустить `vi` — интересно, вы догадаетесь, что делать дальше? Думаю, нет.

Редактор `vi` создавался в 70-е годы прошлого века — тогда ни о каком комфорте и удобном интерфейсе пользователя речи быть не могло, однако на тот момент программа была настоящим прорывом вперед. Вся беда `vi` в том, что он остался примерно на том же уровне — на уровне 70-х годов и совсем не годится для современного пользователя. За эти слова меня могут ругать гуру UNIX, но совершенно не вижу смысла использовать неудобный `vi`, изучать и запоминать его команды, если можно просто запустить другой редактор — современный и удобный. Если есть огромное желание, можете изучить страницу руководства (`man vi`), но я рекомендую использовать один из следующих редакторов: `nano`, `pico`, `mcedit`. Последний входит в состав файлового менеджера Midnight Commander (пакет называется `mc`), который как две капли воды похож на всем известный Norton Commander. В вашей системе редакторы `nano` и `mcedit` могут быть не установлены, для их установки требуются пакеты `mc` и `nano`. Следующие команды произведут инсталляцию этих пакетов в openSUSE и Ubuntu:

```
sudo zypper install mc nano
sudo apt-get install mc nano
```

Редактор `mcedit` мне нравится даже больше, чем `nano`, поскольку он обладает подсветкой синтаксиса, что немаловажно при программировании. Позже мы познако-

мимся с графическими редакторами, но пока в этом нет такой необходимости. Если вы работаете в графическом режиме, вы можете запустить эти редакторы в эмуляторе терминала:

```
mcedit <имя файла>
nano <имя файла>
```

5.2. Компилятор *gcc*

5.2.1. Установка компилятора

Традиционно процесс компиляции состоит из двух этапов: трансляции исходного кода программы на язык, близкий к машинному, и последующей компоновке (некоторые программисты употребляют термин "линковка" — от англ. *link*). Ранее для получения исполнимого кода программы (в DOS/Windows — exe/com-файла) использовались две программы — транслятор, на выходе которого мы получали объектный файл (с расширением obj), и компоновщик (linker), создающий из объектного файла исполнимый файл.

Современные компиляторы совмещают оба эти этапа, и на выходе мы сразу получаем исполнимый код программы. Конечно, любой современный компилятор может создать и объектный файл, если вам это нужно.

Раньше Linux была операционной системой для программистов, использовали ее в основном фанаты, умеющие программировать по прямому назначению — создавали различные программы, совершенствовали код самой операционной системы. Затем Linux благодаря своим корням (как ни крути, а это UNIX-подобная система) превратилась в отличную серверную платформу. Сейчас же разработчики всех дистрибутивов стараются охватить еще одну нишу — пользовательские компьютеры. И это неплохо получается, да так, что инструменты программиста даже не устанавливаются на ваш компьютер. Логика понятна: большинству пользователей нужен офисный пакет OpenOffice.org, но не нужен компилятор `gcc`. Именно поэтому все необходимые инструменты программиста нужно устанавливать вручную. Приведу список пакетов, необходимых нам на данный момент (в процессе чтения книги нужно будет установить дополнительные пакеты):

- `gcc` — компилятор C (GNU C Compiler);
- `gcc-c++` — компилятор `gcc` для C++;
- `binutils` — дополнительные инструменты программиста (`gprof`, `as`, `ld`, `objcopy`, `strings` и др.);
- `make`, `automake` — программы для обработки и создания Makefile (специальных файлов, содержащих опции компилятора для сложных проектов);
- `linux-kernel-headers` — заголовочные файлы ядра (даже если вы не планируете создавать модули ядра, эти файлы пригодятся вам в ваших программах), обычно этот пакет устанавливается при установке `gcc`;

```
den@dhsilabs:~> sudo zypper install gcc
root's password:
Получение метаданных репозитория 'Libdvdcss repository' [готово]
Сбор кэша репозитория 'Libdvdcss repository' [готово]
Получение метаданных репозитория 'Packman Repository' [готово]
Сбор кэша репозитория 'Packman Repository' [готово]
Получение метаданных репозитория 'openSUSE-11.2-Update' [готово]
Сбор кэша репозитория 'openSUSE-11.2-Update' [готово]
Загрузка данных о репозиториях...
Чтение установленных пакетов...
Разрешение зависимостей пакетов...

Будут установлены следующие НОВЫЕ пакеты:
  gcc gcc44 glibc-devel linux-kernel-headers

4 новых пакета для установки.
Полный размер загрузки: 7,9 MiB. После этой операции будет использовано
дополнительно 28,7 MiB.
Продолжить? [y/n/?] (y):
```

Рис. 5.1. Команда `sudo zypper install gcc`

- `glibc-devel` — набор библиотек, необходимый для разработки приложений на C, обычно этот пакет также устанавливается при установке `gcc`.

На рис. 5.1 приведена установка `gcc` в OpenSUSE.

5.2.2. Компиляция первой программы в Linux

Вашей первой программой на языке с вероятностью 99% была программа, выводящая строку "Hello, world!" Такова традиция и не будем ее нарушать. В листинге 5.1 приводится та самая программа, известная всем программистам.

Листинг 5.1. Программа Hello, world! (файл hello.c)

```
#include <stdio.h>

int main(void) {
       printf("Hello, world!");
       return 0;
}
```

На рис. 5.2 изображена эта программа в редакторе `mcedit`. Обратите внимание на подсветку синтаксиса (правда, не знаю, как будет выглядеть подсветка на черно-белой иллюстрации, самый лучший способ оценить подсветку — запустить `mcedit` и увидеть все своими глазами).

Теперь запустим компилятор:

```
gcc hello.c
```

Если вы не умудрились допустить в этой простейшей программе ошибку, то вы ничего не увидите — через секунду компилятор завершит работу. А в текущем ката-

логе появится исполнимый файл a.out. Это и есть ваша скомпилированная программа, запустите ее:

```
./a.out
```

Вы увидите заветную строчку. Поздравляю! Вы только что создали свое первое Linux-приложение!

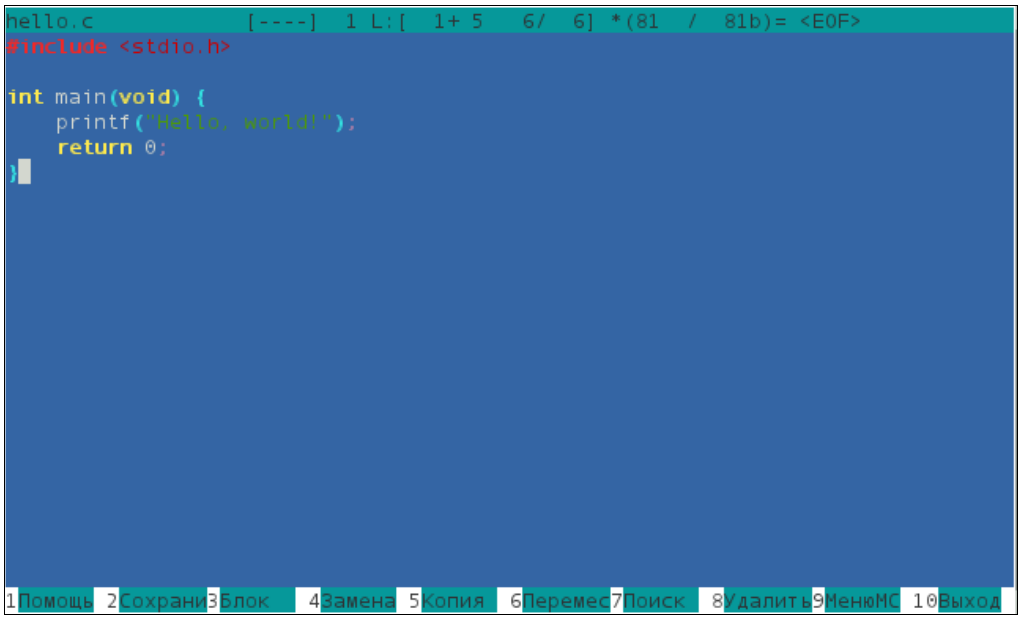


Рис. 5.2. Редактор `mcedit`

Примечание

Если же вы-таки допустили ошибку, то компилятор выведет имя файла, в котором она допущена, номер строки с ошибкой и краткое описание самой ошибки.

5.2.3. Опции компилятора

Общий формат вызова компилятора `gcc` следующий:

```
gcc [параметры] файл
```

Параметров у `gcc` очень много, помнить все из них вам не нужно — в любой момент вы можете прочитать назначение того или иного параметра в справочной системе (команда `man gcc`). Один из наиболее полезных для нас в данный момент параметров — параметр `-o`, задающий имя исполнимого файла, поскольку имя a.out мало кому понравится. Параметр `-v` выводит дополнительную информацию в процессе компиляции. Давайте перекомпилируем нашу программу, чтобы увидеть эту информацию, заодно продемонстрируем использование параметра `-o`:

```
gcc -v -o hello hello.c
```

Параметр `-v` выведет много информации (рис. 5.3) — начиная от расположения заголовочных файлов и версии компилятора до списка используемых библиотек. Даже для такой простой программы опция `-v` сгенерирует много текста: процесс компиляции вовсе не такой простой, как кажется на первый взгляд.

Опция `-o` создаст исполнимый файл hello вместо a.out, что намного удобнее. Запустить hello можно аналогично:

```
./hello
```

```
den@dhsilabs:~/module> gcc -v -o hello hello.c
Using built-in specs.
Target: i586-suse-linux
Configured with: ../configure --prefix=/usr --infodir=/usr/share/info --mandir=/
usr/share/man --libdir=/usr/lib --libexecdir=/usr/lib --enable-languages=c,c++,o
bjc,fortran,obj-c++,java,ada --enable-checking=release --with-gxx-include-dir=/u
sr/include/c++/4.4 --enable-ssp --disable-libssp --with-bugurl=http://bugs.opens
use.org/ --with-pkgversion='SUSE Linux' --disable-libgcj --disable-libmudflap --
with-slibdir=/lib --with-system-zlib --enable-__cxa_atexit --enable-libstdcxx-al
locator=new --disable-libstdcxx-pch --enable-version-specific-runtime-libs --pro
gram-suffix=-4.4 --enable-linux-futex --without-system-libunwind --with-arch-32=
i586 --with-tune=generic --build=i586-suse-linux
Thread model: posix
gcc version 4.4.1 [gcc-4_4-branch revision 150839] (SUSE Linux)
COLLECT_GCC_OPTIONS='-v' '-o' 'hello' '-mtune=generic' '-march=i586'
 /usr/lib/gcc/i586-suse-linux/4.4/cc1 -quiet -v hello.c -quiet -dumpbase hello.c
 -mtune=generic -march=i586 -auxbase hello -version -o /tmp/cchKgKUR.s
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc/i586-suse-linux/4.4/include
 /usr/lib/gcc/i586-suse-linux/4.4/include-fixed
 /usr/lib/gcc/i586-suse-linux/4.4/../../../../i586-suse-linux/include
 /usr/include
End of search list.
GNU C (SUSE Linux) version 4.4.1 [gcc-4_4-branch revision 150839] (i586-suse-lin
ux)
        compiled by GNU C version 4.4.1 [gcc-4_4-branch revision 150839], GMP ve
rsion 4.3.1, MPFR version 2.4.1-p5.
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: af2df01b73873daf15546c72f572628c
COLLECT_GCC_OPTIONS='-v' '-o' 'hello' '-mtune=generic' '-march=i586'
```

Рис. 5.3. Вывод параметра `-v`

В следующей главе мы поговорим о библиотеках C, а сейчас рассмотрим несколько опций компилятора, относящихся к библиотекам. Опция `-l` позволяет указать имя библиотеки, которую нужно использовать при сборке программы:

```
gcc -ldlib file.c
```

Название библиотеки задается сразу после параметра `-l` — без пробелов. Компоновщик (часть `gcc`) будет искать файл libdlib.a в системных каталогах библиотек и каталогах, которые вы укажете с помощью опции `-L`. Обратите внимание: параметр `-l` (маленькая "L") задает название библиотеки, а параметр `-L` — список каталогов, в которых следует искать эту библиотеку.

Опция -nostdlib запрещает использовать системные библиотеки, будут использоваться только те, которые компоновщик найдет в каталогах, заданных параметром -L:

```
gcc -o hello hello.c -L/usr/lib/my -lmylib
```

Параметр -static задает статическую компоновку. Подробно все параметры, относящиеся к библиотекам, будут рассмотрены в *главе 6*.

Параметр -I задает каталог с заголовочными файлами. По умолчанию используются системные каталоги для заголовочных файлов, однако если вы желаете подключить собственный каталог с заголовочными файлами, это можно сделать с помощью параметра -I:

```
gcc -I/home/den/include program.c
```

Чтобы добавить отладочную информацию в исполнимый файл, нужно использовать опцию -g. Отладочная информация увеличивает размер исполнимого файла, но позволяет отлаживать программу отладчиком gdb, который будет рассмотрен в *главе 34*.

Когда программа отлажена и все логические ошибки (они же "баги") исправлены, тогда нужно перекомпилировать программу без параметра -g. При этом можно еще задать опции оптимизации, позволяющие уменьшить размер исполнимого файла и ускорить запуск программы. Опция -O включает базовую оптимизацию (уровень 1), но вы можете использовать опции -O1, -O2, -O3 — поэкспериментируйте с ними. Обратите внимание на то, как запускается программа, и на размер исполнимого файла. Обычно опция -O1 (то же что и -O) позволяет уменьшить размер исполнимого файла и ускорить запуск программы. Опция -O2 уменьшит размер файла до минимума. Уровень 3 (-O3) — максимальная оптимизация.

На самом деле разница между опциями -O1, -O2 и -O3 небольшая. Разница заметна между опциями -O0 (отключение оптимизации) и -O1: программа, скомпилированная с -O1, будет выполняться в 5–10 раз быстрее, чем программа без оптимизации. В *главе 35* будет рассмотрен профайлер gprof, позволяющий вычислить время выполнения программы и всех ее функций. Используйте его для экспериментов с опциями оптимизации. Если вам нужна дополнительная информация (в том числе приводится время выполнения тестовой программы, скомпилированной с разными опциями оптимизации), вы можете почерпнуть ее из статьи "Изучаем параметры gcc":

http://gentoo.theserverside.ru/book/ar72.html.

5.3. Автоматическая сборка программ

5.3.1. Введение в автоматическую сборку

При сборке программы нужно указывать много параметров компилятора в командной строке. В результате получается длинная команда, набирая которую каждый раз после исправления очередной ошибки, можно легко допустить ошибку. Да, есть

история командной строки, но это не панацея. А что, если вам нужно пересобрать сложный проект, состоящий из десятка объектных файлов? Понадобится уже не одна такая команда.

Первое, что приходит в голову после прочтения первой части книги, — использование сценариев командной оболочки. Что ж, решение вполне нормальное, если бы не одно "но". Незачем изобретать велосипед, когда он уже существует. В Linux (и других UNIX-системах) принято использовать для сборки сложных проектов утилиту `make`. Суть заключается в следующем: вы создаете так называемый Makefile, в котором указываются цели для сборки (программы, библиотеки, объектного файла). Файл Makefile находится в одном с вашим проектом каталоге. Для сборки всего проекта достаточно ввести команду:

```
make
```

Программа `make` прочитает Makefile и выполнит указанные в нем действия. А действия могут быть самые разные, например вы можете создать цель clean, которая ничего не будет компилировать, зато будет "убирать" мусор, например удалять объектные файлы, которые уже не нужны. Пусть ваша программа состоит из нескольких объектных файлов. После компиляции самой программы эти объектные файлы уже не нужны, но продолжают занимать место на диске. Цель clean удалит эти файлы.

Программа `make` — стандарт в мире Linux. Конечно, при разработке *вашей* программы вы можете использовать любые средства, но если вы планируете распространять исходный код программы, что вполне естественно для Linux-приложений, то все сообщество Linux-программистов будет считать вас чудаком, если в вашем проекте не будет Makefile. Отказ от Makefile будет расценен в лучшем случае как просто странность, в худшем — как непрофессионализм.

Кроме `make` можно использовать и другие средства автоматической сборки, например программы `automake` и `autoconf`. В этой книге данные программы не рассматриваются, поскольку обычно программисты отдают предпочтение классической программе `make`.

Алгоритм разработки программы при использовании `make` обычно следующий: после подготовки исходного кода создается Makefile и выполняется команда `make`. Если собранная программа не содержит ошибок и нормально выполняется, то запускается цель make, удаляющая все лишнее, а иногда и даже скомпилированную программу. Делается это для того, чтобы можно было запаковать в архив исходный код программы с целью его распространения. Конечно, можно распространять исходный код вместе с скомпилированной программой, но, как правило, исполняемые файлы проекта распространяются отдельно от исходного кода проекта.

5.3.2. Синтаксис Makefile

Основное содержимое Makefile — цели или целевые связки, их синтаксис будет рассмотрен чуть позже, а пока поговорим о необязательных элементах файла сборки.

Для сложных проектов неплохо бы описать, для чего используется та или иная цель — прокомментировать записи файла сборки. Комментарии в файле сборки начинаются с символа решетки, допускаются только однострочные комментарии — каждая строка комментария должна начинаться с решетки:

```
# Цель clean
# Удаляет объектные файлы
clean:
	rm -f *.o
```

Кроме комментариев в файле сборки могут быть константы, которые будут использоваться для подстановки, например:

```
CC=gcc
...
obj1.o: obj1.c
	$(CC) -c obj1.c
```

В конечном итоге, если вы надумаете использовать другой компилятор для сборки программы, достаточно изменить константу `CC`, а не производить замену во всем файле сборки.

Теперь приступим к написанию целей или целевых связей — основного содержимого Makefile. Первым делом нужно объявить основную цель, затем — промежуточные цели. Рассмотрим Makefile (листинг 5.2), используемый для сборки библиотеки (сам процесс создания библиотеки будет рассмотрен в следующей главе).

Листинг 5.2. Makefile для сборки библиотеки

```
libmy.so: obj1.o obj2.o
	gcc -shared -o libmy.so $^
obj1.o: obj1.c
	gcc -fPIC -c $^
obj2.o: obj2.c
	gcc -fPIC -c $^
clean:
	rm -f libmy.so *.o
```

Примечание

Опция `-c` указывает компилятору не запускать компоновщик (линкер), в результате будет создан объектный код, а не исполнимый файл. Опция `-c` используется обычно для получения объектных файлов, которые потом будут объединены в одну библиотеку.

В нашем случае цель libmy.so является основной, она требует сборки дополнительных целей obj1.o и obj2.o. Цель clean выбивается из общей цепочки, она не будет выполнена при введении команды `make`. Чтобы выполнить цель clean, нужно ввести команду:

```
make clean
```

Целевая связка состоит из следующих компонентов:

- имя цели — в нашем случае именем главной цели служит libmy.so, поскольку цель собирает библиотеку libmy.so, но вы можете назвать цель как вам угодно — название цели не влияет на имя результирующего файла. Имя результирующего файла, как обычно, задается опцией `-o` компилятора `gcc`. Учитывая, что libmy.so — главная цель, мы могли бы назвать ее main;
- список зависимостей — после имени цели следует двоеточие, после которого перечисляются вспомогательные цели, в нашем случае — obj1.o и obj2.o. Цели obj1.o и obj2.o должны быть собраны прежде, чем будет собрана наша главная цель. Главная цель зависит от дополнительных целей. Если зависимостей у цели нет, то они не указываются (см. цель clean);
- команды — после списка зависимостей следуют команды, которые нужно выполнить для достижения цели. Команды могут быть любыми — не обязательно вызывать `gcc`. Например, для достижения цели clean выполняется команда `rm`, а в *главе 6* вы увидите, как запускается архиватор `ar`. Возможность запуска любых команд делает команду `make` более универсальной. Каждая команда начинается с символа табуляции и пишется с новой строки.

> ***Внимание!***
>
> Некоторые текстовые редакторы заменяют символ табуляции определенным числом пробелов. Такой Makefile работать не будет! Нужно или сменить редактор, или настроить его так, чтобы в файл вносился именно символ табуляции (`\t`), а не восемь пробелов.

Еще раз посмотрите на листинг 5.2. Вас сбивают с толку имена целей, поскольку они совпадают с именем файла? Это не проблема — давайте немного усовершенствуем Makefile, добавив в него константы и комментарии (листинг 5.3).

Листинг 5.3. Усовершенствованный Makefile

```
CC=gcc
# Главная цель
main: first_object second_object
	$(CC) -shared -o libmy.so obj1.o obj2.o
# Цель для сборки файла obj1.c
first_object: obj1.c
	$(CC) -fPIC -c $^
# Цель для сборки файла obj2.c
second_object: obj2.c
	$(CC) -fPIC -c $^
# Очистка проекта
clean:
	rm -f libmy.so *.o
```

Теперь все стало на свои места. Главная цель называется main, для ее сборки нужно сначала собрать дополнительные цели first_object и second_object. Все эти цели вызывают компилятор, указанный константой `CC`.

Кроме констант, определенных пользователем, `make` позволяет использовать две стандартные константы:

- `$^` — содержит список всех зависимостей текущей цели;
- `$@` — содержит имя текущей цели.

Посмотрите на нашу цель main:

```
main: first_object second_object
       $(CC) -shared -o libmy.so obj1.o obj2.o
```

Раньше она называлась libmy.so и выглядела так:

```
libmy.so: obj1.o obj2.o
       gcc -shared -o libmy.so $^
```

Сравните команды обеих целей. Во втором случае (libmy.so) мы использовали подстановку `$^`, чтобы передать компилятору список объектных файлов, имена которых совпадали с именами целей `make`. Именно поэтому в листинге 5.2 мы использовали имена целей, совпадающие с именами результирующих файлов.

Цель libmy.so из листинга 5.2 можно было еще сократить так:

```
libmy.so: obj1.o obj2.o
       gcc -shared -o $@ $^
```

Вроде бы с целями разобрались, настало время поговорить о запуске `make`. Программе `make` нужно передать список целей, которые нужно выполнить, например:

```
make main
```

В данном случае будет выполнена цель main и будет собран текущий проект. Название цели main должно быть указано в файле Makefile, находящемся в одном каталоге с проектом.

Вы можете создать несколько файлов Makefile — для разных вариантов сборки программы. Например, Makefile — для сборки программы обычным компилятором `gcc`, а Makefile.intel — для сборки программы компилятором компании Intel. Указать программе `make`, какой файл нужно использовать, можно опцией `-f`:

```
make -f Makefile.intel
```

> ***Примечание***
>
> Познакомиться с компилятором C++ компании Intel для ОС Linux можно по адресу:
>
> **http://software.intel.com/en-us/articles/intel-c-compiler-professional-edition-for-linux-documentation/**.

При вызове `make` цель можно вообще не указывать. В этом случае основной целью будет считаться первая цель в Makefile.

Еще раз посмотрите на листинг 5.2 (или 5.3 — не имеет значения). После того как вы введете команду `make`, должны быть выполнены следующие команды:

```
gcc -fPIC -c obj1.c
gcc -fPIC -c obj2.c
gcc -shared -o libmy.so obj1.o obj2.0
```

Команда `rm -f libmy.so *.o` будет выполнена, только если мы явно укажем цель clean:

```
make clean
```

Итак, в этой главе мы познакомились не только с компилятором `gcc`, но и с утилитой автоматической сборки проекта — программой `make`. В следующей главе мы поговорим о библиотеках — без них не обходится ни один серьезный проект.

ГЛАВА 6

Библиотеки. Автоматическая сборка библиотек

6.1. Динамические и статические библиотеки

Пользователи, ранее программировавшие на C, наверняка уже сталкивались с библиотеками — без них не обходится ни один серьезный проект. Библиотеки представляют собой набор соединенных объектных файлов, что позволяет разным программам использовать один и тот же объектный код, что в итоге существенно сокращает размер конечной программы. Представьте, что у нас есть библиотека для создания графического интерфейса "весом" в 10 Мбайт. Когда вы напишете свою программу, использующую эту библиотеку, размер исполнимого файла (в зависимости от сложности самой программы) составит, допустим, 1 Мбайт.

Если же весь код для организации графического интерфейса включить в состав программы, то размер ее исполнимого файла будет 11 Мбайт. Разница начинает ощущаться, когда таких программ не одна, а 10 или даже 50. Вместо 21 Мбайт эти 10 программ займут 110 Мбайт.

Библиотеки подключаются к программе на стадии компоновки. Например, та же функция `printf()` реализована в стандартной библиотеке языка C, которая автоматически подключается к программе компилятором `gcc`.

Чтобы подключить к программе дополнительную библиотеку, нужно использовать опцию `-l`, например:

```
gcc -o test test.c -lmylib
```

Обычно файлы библиотек находятся в каталогах /lib, /usr/lib, /usr/local/lib. Если же подключаемая библиотека находится в другом каталоге, то он указывается опцией `-L`:

```
gcc -o test test.c -L/usr/lib/my -lmylib
```

Имена файлов библиотек начинаются с префикса lib, т. е., когда вы подключаете библиотеку mylib, ее файл на диске будет называться libmylib.so (или libmylib.a — в зависимости от типа библиотеки).

Библиотеки могут быть статическими или динамическими (их еще называют совместно используемые). Статические библиотеки создаются программой `ar`, "рас-

ширение" файла (напомню, что в Linux нет понятия "расширение файла", но так будет понятнее читателю) у статических библиотек — a (например, libmylib.a). Если вы создали библиотеку libmylib.a и поместили в нее, к примеру, три объектных файла obj1.o, obj2.o, obj3.o, то вы можете либо использовать эту библиотеку, либо указать объектные файлы непосредственно в командной строке вызова компилятора:

```
gcc -o test test.c -lmylib
gcc -o test test.c obj1.o obj2.o obj3.o
```

Эти две команды полностью аналогичны, но, как правило, использование библиотек, даже статических, намного удобнее, поскольку не нужно перечислять в командной строке все необходимые программе объектные файлы.

Динамические библиотеки создаются самим `gcc` — когда он запущен с опцией `-shared`. "Расширение" файла у динамических библиотек — so. Разница между статическими и динамическими (совместно используемыми) библиотеками — огромная. И дело не в том, что первые создаются архиватором, а вторые — компилятором. Важен сам механизм их работы. Статическая библиотека при компиляции программы полностью встраивается в нее. Если три наших файла obj1.o, obj2.o, obj3.o занимают, скажем, 20 Мбайт, то наша программа станет на эти 20 Мбайт "тяжелее", поскольку объектный код полностью будет внедрен в исполнимый файл программы при компиляции.

При использовании динамических библиотек объектный код не встраивается в исполнимый файл программы, вместо этого создается ссылка на этот код, находящийся в отдельном файле — в файле библиотеки. Поэтому если программа использует какую-нибудь библиотеку, для ее запуска в системе должна быть установлена эта библиотека, иначе программа просто не запустится.

Как вы уже догадались, поскольку статические библиотеки встраиваются в исполнимый файл, то наличие самой библиотеки на целевой системе не обязательно. Статические библиотеки создаются исключительно для удобства программиста — не нужно переносить один и тот же код при создании разных программ. А динамические библиотеки не только упрощают труд программиста, но и позволяют экономить место на диске.

> ***Примечание***
>
> Некоторые не особо опытные программисты считают библиотеками заголовочные файлы, однако это не так. Заголовочные файлы — это всего лишь часть программного (исходного) кода программы, они не являются объектным кодом, следовательно, не могут называться библиотеками.

Некоторые библиотеки в системе представлены как в статическом, так и динамическом вариантах (присутствуют файлы как с "расширением" so, так и с "расширением" a).

Какой вариант выберет компилятор при сборке вашей программы? По умолчанию предпочтение отдается динамическому варианту библиотеки, а статический используется, только если указана опция `-static`, например:

```
gcc -static -o test test.c -lm
```

Помните, что размер исполнимого файла при статической компоновке будет существенно выше, поэтому используйте параметр `-static`, только если вы действительно в нем нуждаетесь.

6.2. Создание статической библиотеки

Статическая библиотека, как уже было отмечено ранее, — это всего лишь архив объектных файлов, созданный программой `ar`. К сожалению, Windows "извратила" даже само слово "архиватор". Подавляющее большинство Windows-пользователей считает, что архиваторы используются для сжатия файлов и экономии места на диске, поскольку само только слово "архиватор" ассоциируется с программами WinZip, WinRAR и им подобными. На самом деле архиватор не выполняет сжатие файлов, он просто объединяет несколько файлов в один таким образом, что возможна обратная процедура (т. е. можно добавить файлы в архив и извлечь их оттуда). Именно такие функции и выполняет программа `ar`, входящая в пакет `binutils` (пакет устанавливается по умолчанию, поэтому не нужно выполнять каких-либо действий для установки программы `ar`). Программы, выполняющие *сжатие* архивного (и не только) файла, называются *компрессорами*. Например, программы `tar` и `ar` — архиваторы, а программы `gzip` и `bzip` — компрессоры. Архиваторы (например, `tar`) могут вызывать компрессоры для получения уже сжатого архива.

Чтобы поместить в архив файлы obj1.o, obj2.o и obj3.o, используется команда:

```
ar r libmy.a obj1.o obj2.o obj3.o
```

При желании можно извлечь файлы из архива, но в случае с библиотекой это вам вряд ли понадобится:

```
ar x libmy.a
```

Теперь давайте напишем простую библиотеку. Создайте два файла — obj1.c (листинг 6.1) и obj2.c (листинг 6.2).

Листинг 6.1. Файл obj1.c

```
#include <stdio.h>
void hello1 (void)
{
       printf("Hello from obj1\n");
}
```

Листинг 6.2. Файл obj2.c

```
#include <stdio.h>
void hello2 (void)
{
       printf("Hello from obj2\n");
}
```

Наши функции предельно просты — их задача продемонстрировать создание библиотеки.

Для сборки нашей библиотеки мы будем использовать программу `make`. Сначала создадим Makefile (листинг 6.3).

Листинг 6.3. Makefile для сборки библиотеки

```
libmy.a: obj1.o obj2.o
	ar r $@ S^
obj1.o: obj1.c
	gcc -c $^
obj2.o: obj2.c
	gcc -c $^
clean:
	rm -f libmy.a *.o
```

Наш Makefile предусматривает четыре цели: цель для создания библиотеки libmy.a, которая запускает программу `ar` и упаковывает в нее файлы obj1.o и obj2.o. В свою очередь, цели obj1.o и obj2.o создают соответствующие объектные файлы (вызывают компилятор `gcc` для сборки исходного кода библиотек). Цель clean удаляет созданную библиотеку и объектные файлы.

Запустим `make` для сборки библиотеки:

```
make
```

Теперь, когда библиотека создана, напишем программу, которая будет использовать эту библиотеку (листинг 6.4).

Листинг 6.4. Программа test (test.c)

```
#include <stdio.h>

int main(void)
{
	hello1();
	hello2();
	return 0;
}
```

Осталось только собрать все воедино:

```
gcc -o test test.c -L. -lmy
```

Компилятор `gcc` будет "собирать" программу test.c в исполнимый файл test с использованием библиотеки `my` (префикс lib и "расширение" файла не указывается), поиск библиотеки будет производиться в текущем каталоге (`-L.`).

Запустим программу test:

```
./test
Hello from obj1
Hello from obj2
```

6.3. Создание динамической библиотеки

Процесс создания динамической библиотеки не так уж и сложен. Разница заключается в том, что используется компилятор `gcc`, а не архиватор `ar`. Представим, что у нас уже есть объектные файлы obj1.o и obj2.o, которые нам нужно включить в состав нашей динамической библиотеки. Для создания библиотеки используется команда:

```
gcc -shared -o libmy.so obj1.o obj2.o
```

В общем случае формат вызова `gcc` будет таким:

```
gcc -shared -o имя_библиотеки список_объектных_файлов
```

По сути, процесс создания библиотеки мало чем отличается от компиляции программы, разница только в обязательном параметре `-shared`.

Но, как обычно, есть одно "но". В самом начале этого раздела я сделал предположение, что объектные файлы у нас уже есть, но я не сказал, как мы их получили. Чтобы собрать объектные файлы в состав динамической библиотеки, они должны быть скомпилированы с опцией `-fPIC`, обеспечивающей получение объектного файла с позиционно-независимым кодом (Position Independent Code). Другими словами, объектные файлы из сборки статической библиотеки использовать нельзя — их придется перекомпилировать.

Однако и это еще не все. Мы уже знакомы с опцией компилятора `-L`, позволяющей дополнить список каталогов, в которых нужно искать библиотеку. Однако эта опция позволяет найти библиотеку только компилятору, но не вашему приложению при последующем запуске.

Сейчас поясню, что имелось в виду. Вы компилируете программу и указываете путь к библиотеке — к каталогу /home/den/mylibs:

```
gcc -o test test.c -L/home/den/mylibs -lmy
```

Понятно, что на другом компьютере каталога /home/den/mylibs не будет. Мало того: даже если вы запустите программу, использующую динамическую библиотеку libmy.so, будет произведен поиск этой библиотеки. В 99% случаев каталог /home/den/mylibs, содержащий библиотеку libmy.so, не будет включен в системный список каталогов библиотек (`LD_LIBRARY_PATH`).

В момент запуска программы производится поиск библиотек в каталогах, перечисленных в файле /etc/ld.so.conf и в переменной окружения `LD_LIBRARY_PATH`.

Что нужно, чтобы библиотека была найдена? Есть несколько способов:

- добавить библиотеку в один из каталогов, перечисленных в /etc/ld.so.conf — проследите, чтобы при установке программы библиотека копировалась в нужный каталог;
- указать нужный вам каталог в /etc/ld.so.conf — способ не очень корректный, поскольку при запуске системы просматривается /etc/ld.so.conf и создается кэш динамических библиотек (файл кэша называется /etc/ld.so.cache). Если каждый

программист будет добавлять в этот файл свой каталог, это отрицательно скажется на запуске всей системы. Обычно в этот файл добавляются каталоги с большими библиотеками — X11, GTK, Qt и т. д.

Независимо от того, какой способ вы выбрали, при компиляции программы нужно указать, где ей искать используемую библиотеку. Это можно сделать с помощью опций `-W1` (указывает передать компоновщику определенную опцию, в нашем случае — `-rpath`) и `-rpath` (указывает путь к библиотеке).

Представим, что вы поместили библиотеку в каталог /usr/lib/my и "прописали" этот каталог в /etc/ld.so.conf. Тогда для сборки программы нужно использовать следующую команду:

```
gcc -o test test.c -L/usr/lib/my -lmy -W1,-rpath,/usr/lib/my
```

Теперь разберемся, что есть что. Компилятор должен скомпилировать программу test.c (имя исполнимого файла — test). Мы указываем компилятору, что программа использует библиотеку my (файл libmy.so), которую нужно искать в каталоге /usr/lib/my, но вместо этого каталога вы можете указать и каталог, в котором "собиралась" библиотека, например /home/den/mylibs, — разницы не будет никакой (главное, чтобы в этом каталоге был файл libmy.so). Опция `-W1` нужна для компоновщика. Она указывает каталог, в котором будет найдена нужная библиотека *при запуске программы*. Обратите внимание на саму опцию `-W1`: после запятой указывается опция, которую нужно передать компоновщику `ld` (в данном случае — `-rpath`), аргументы опции также указываются через запятую (аргумент `-rpath` — имя каталога с библиотекой).

Чтобы не усложнять себе жизнь лишними опциями, лучше всего размещать библиотеки в каталогах, уже указанных в ld.so.conf, например в /usr/lib или /usr/local/lib, — тогда для сборки программы можно использовать команду:

```
gcc -o test test.c -lmy
```

Как видите, так существенно проще.

> ***Примечание***
>
> Сразу после помещения в один из каталогов, указанных в ld.so.conf, вашей библиотеки система не "увидит" ее. Для обновления кэша ld.so.cache без перезагрузки системы используется утилита `ldconfig`, запускать которую нужно с правами root. Подробности вы можете прочитать в `man ldconfig`.

Нам осталось создать Makefile, облегчающий сборку нашей библиотеки (листинг 6.5).

Листинг 6.5. Makefile для динамической библиотеки

```
libmy.so: obj1.o obj2.o
	gcc -shared -o libmy.so $^
obj1.o: obj1.c
	gcc -fPIC -c $^
```

```
obj2.o: obj2.c
	gcc -fPIC -c $^
clean:
	rm -f libmy.so *.o
```

Сравните этот Makefile с аналогичным, но для статической библиотеки (см. листинг 6.3). Разница заметна сразу:

- библиотека называется libmy.so, а не libmy.a;
- для сборки библиотеки вызывается `gcc` с опцией `-shared`, а не `ar`;
- сборка объектных файлов осуществляется с опцией `-fPIC`.

Если подытожить наши знания о статических и динамических библиотеках, то получается вот что:

- статические библиотеки проще в использовании для программиста: вам не нужно беспокоиться, найдет ли программа библиотеку при запуске — ведь объектный код библиотеки будет "встроен" в исполнимый код программы;
- динамические библиотеки сложнее в использовании — нужно все предусмотреть: найдет ли программа библиотеку при запуске, будет ли правильно размещена библиотека при установке программы, будет ли обновлен кэш библиотек и т. д;
- зато динамические библиотеки позволяют сэкономить дисковое пространство.

ГЛАВА 7

Переменные окружения

7.1. Еще один способ передачи параметров

Какие мы знаем способы передачи параметров программы? Первое, что приходит на ум, — это командная строка. Да, так и передавались параметры программам с давних времен. Но, согласитесь, когда вы часто запускаете программу с одним и тем же набором параметров, то не очень хочется вводить их каждый раз при запуске программы. Где бы их сохранить? Можно хранить параметры программы в конфигурационных файлах. В Linux общесистемные параметры, если программа подразумевает использование таких параметров, как правило, хранятся в каталоге /etc. Пользовательские параметры хранятся обычно в домашнем каталоге пользователя. Подкаталог и имя файла зависят от разработчика и уникальны для каждой программы.

Есть еще один способ передачи параметров — с помощью окружения. Переменные окружения можно установить не только в Linux, но и в Windows, однако в Windows этот способ передачи параметров используется крайне редко, а вот в UNIX-подобных системах, наоборот, используется часто.

Чтобы понять, как работают переменные окружения, нам сначала нужно разобраться, что такое окружение. Об этом и поговорим в следующем разделе.

7.2. Что такое окружение?

Концепцию многозадачности мы еще не рассматривали, но немного забежим наперед. Все мы знаем, что такое процесс (ну, или, во всяком случае, догадываемся). Будем считать процессом запущенную программу. Любой процесс может запустить другую программу — она будет считаться дочерним процессом. Например, после входа в систему автоматически запускается командный интерпретатор `bash`, принимающий ваши команды. Когда вы введете команду `mc`, запустится файловый менеджер Midnight Commander, который будет дочерним процессом (или потомком — так проще) по отношению к `bash`. В свою очередь, `bash` будет родительским процессом (или родителем).

Все процессы в системе являются элементами одной иерархии, т. к. ничего так просто не запускается — всегда находится процесс, который запустит следующий процесс. На вершине иерархии программа `init` (да, она играет роль системы инициализации Linux во многих дистрибутивах). В Linux программа `init` — единственный процесс без родителя (программа `init` запускается непосредственно ядром при запуске системы). Кстати, у ядра есть параметр `init`, с помощью которого можно изменить систему инициализации — тогда вместо `init` будет запущена другая программа.

Окружение — это набор пар вида ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, окружение уникально для каждого пользователя, например у вас может быть один набор пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, а у другого пользователя — другой набор. Одинаковым для двух пользователей окружение быть не может, т. к. даже если совпадает список переменных, их значения будут другими — как минимум, будет другое значение у переменных окружения `USER` (содержит текущее имя пользователя) и `HOME` (содержит домашний каталог).

Для просмотра окружения вашей оболочки введите команду `env`. Вы увидите множество переменных, но на практике часто используются следующие:

- `USER` — содержит имя текущего пользователя (пользователя, под которым вы зашли в систему);
- `HOME` — содержит имя домашнего каталога пользователя;
- `PWD` — текущий (рабочий) каталог;
- `SHELL` — оболочка пользователя (обычно `bash`);
- `PATH` — путь поиска исполнимых файлов.

Просмотреть значение переменной можно так:

```
echo $имя
```

Для установки значения переменной используется конструкция:

```
имя=значение
```

Вот небольшой пример:

```
FIRSTPRG=hello
echo $FIRSTPRG
```

Однако сейчас наша переменная не является переменной окружения, она просто переменная оболочки. Чтобы она стала переменной окружения, нужно ее экспортировать:

```
export FIRSTPRG
```

Можно сразу устанавливать значение и экспортировать переменную:

```
export FIRSTPRG=hello
```

После того как наша переменная стала переменной окружения (после экспорта), она становится доступной во всех дочерних процессах. Если вы в командной оболочке устанавливаете переменную окружения, а затем запускаете свою программу,

то ваша программа может "добраться" к переменным окружения оболочки. Переменные окружения можно установить в конфигурационных файлах оболочки (как это сделать, было показано в *главах 2* и *3*). Да, вы на правильном пути: можно существенно сократить код программы, отказавшись от конфигурационных файлов в пользу переменных окружения. Для этого просто в документации следует указать, что необходимо установить такие-то переменные окружения. Может это и некрасиво, и несерьезно, но вам не нужно будет заниматься разбором вашего конфигурационного файла — ведь его не просто нужно прочитать, а "разобрать по косточкам". А используя переменные окружения, вы можете существенно упростить код и сократить время разработки.

7.3. Чтение переменных окружения в вашей программе

Прочитать переменные окружения в вашей программе можно двумя способами. Первый заключается в использовании внешней переменной `environ`, которая объявлена в заголовочном файле unistd.h. Второй подразумевает использование функции `getenv()`.

Я считаю, намного удобнее использовать функцию `getenv()`. Сейчас поясню почему. Переменная `environ` содержит все окружение, т. е. все пары ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, но разбором окружения (т. е. поиском нужных вам переменных с целью определения их значений) придется заниматься вручную, что несколько усложняет код программы. Тогда уж проще создать конфигурационный файл и хранить параметры в нем.

А функция `getenv()` позволяет получить значение определенной переменной окружения. Прототип функции следующий:

```
char * getenv (char * VAR_NAME);
```

Рассмотрим небольшую программу, выводящую имя текущего пользователя (листинг 7.1).

Листинг 7.1. Программа user.c

```
#include <stdio.h>

int main (void)
{
        value = (char*) getenv("USER");
        printf("Текущий пользователь: %s\n", value);
        return 0;
}
```

Программа предельно проста: сначала она получает значение переменной окружения `USER` в переменную `value`, а потом выводит последнюю.

7.4. Модификация окружения

Для изменения переменных окружения используются функции:

- `setenv()` — изменяет существующую или добавляет новую переменную окружения;
- `putenv()` — то же, что и `setenv()`, но для установки переменной окружения использует строку вида ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ;
- `unsetenv()` — удаляет переменную окружения.

Все эти функции объявлены в заголовочном файле stdlib.h. Теперь давайте рассмотрим каждую из этих функций. Начнем с функции `setenv()`:

```
int setenv (const char * NAME, const char * VALUE, int OV);
```

Как вы уже догадались, функция создает (или изменяет значение) переменную с именем `NAME`, значение этой переменной задано параметром `VALUE`. Флаг `OV` — это флаг перезаписи. Если он равен 0, то существующая переменная `NAME` не будет перезаписана, а если содержит значение, отличное от 0, переменной `NAME` будет присвоено новое значение `VALUE`.

Функция `setenv()` возвращает 0, если установка переменной окружения прошла успешно, и –1 в случае ошибки.

Функцию `putenv()` удобно использовать, если у нас есть строка вида ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ и мы не хотим заниматься ее разбором (чтением первой части строки в переменную, которая потом будет передана как параметр `NAME`, и чтением второй части строки, которая станет потом значением новой переменной — параметром `VALUE` функции `setenv()`). Прототип функции:

```
int putenv (char * INITSTR);
```

Пример использования функции:

```
result = putenv("DIR=/etc");
```

Как и функция `setenv()`, данная функция возвращает –1 в случае ошибки и 0, если все прошло успешно. Помните, что функция `putenv()` есть не во всех UNIX-подобных системах, что в конечном итоге может сказаться на переносимости вашей программы.

Функция `unsetenv()` удаляет переменную с именем `NAME` из окружения:

```
int unsetenv(const char * NAME);
```

Возвращаемые значения такие же, как у предыдущих функций. Теперь рассмотрим пример использования этих функций (листинг 7.2).

Листинг 7.2. Модификация окружения

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main (void)
{
        putenv("DIR=/etc");
        setenv ("DIR", "/home/den", 1);

        value = (char*) getenv("DIR");
        printf("DIR = %s\n", value);

        unsetenv("DIR");
        return 0;

}
```

Сначала мы функцией `putenv()` устанавливаем переменную `DIR` (значение `/etc`). Затем мы модифицируем эту переменную (обратите внимание на флаг `OV`) функцией `setenv()`. Потом мы получаем значение переменной с помощью функции `getenv()` и выводим ее. Функция `unsetenv()` удаляет переменную окружения.

Сейчас мы рассмотрим еще одну функцию, которая на практике используется чрезвычайно редко. Более того, она реализована не во всех UNIX-системах, поэтому может у вас не работать:

```
int clearenv(void);
```

Данная функция очищает окружение и делает это максимально корректно, т. е. путем освобождения памяти, используемой для хранения переменной окружения.

Очистить окружение можно и другим путем — просто присвоить переменной `environ` значение `NULL`, но в этом случае мы лишь переустановим указатель, но сами данные останутся в памяти, и при желании к ним можно будет добраться. Функцию `clearenv()` целесообразно использовать на системах, требовательных к безопасности.

ГЛАВА 8

Ввод/вывод в Linux

8.1. Понятие ввода/вывода. Перенаправление ввода/вывода в командной строке

Ввод/вывод (Input/Output, I/O) — понятие довольно абстрактное. В общем, вводом/выводом считается взаимодействие обработчика информации (в нашем случае им будет программа) с внешним миром (с операционной системой и другими программами). Дабы не усложнять себе жизнь, в этой книге мы будем рассматривать только файловый ввод/вывод.

Реализовать ввод/вывод в программе можно двумя способами: использовать библиотечные функции C и использовать системные вызовы Linux. У каждого способа есть свои преимущества. Программа, использующая библиотечные функции, может быть легко портирована в другую систему, где есть компилятор C. Нужно перенести ее исходный код и скомпилировать — в результате у вас появится Windows-версия. Совсем другое дело — программа, использующая системные вызовы. Такая программа читает и записывает файлы, обращаясь к ядру Linux. В этом случае ввод/вывод называется низкоуровневым. Программа будет работать быстрее, чем аналогичная, использующая системные вызовы (это особенно будет заметно при обработке больших объемов информации), но не может быть портирована в другую операционную систему, т. к. там нет ядра Linux, следовательно, системные вызовы работать не будут.

В этой книге мы рассмотрим оба способа организации ввода/вывода, но прежде нам нужно разобраться с вводом/выводом в командной строке. Если вы уже состоялись, как Linux-пользователь, приведенная далее информация вам уже знакома, поэтому можете смело перейти к следующему разделу.

С помощью перенаправления ввода/вывода мы можем перенаправить вывод одной программы в файл или на стандартный ввод другой программы. Например, у вас не получается настроить сеть и вы хотите перенаправить вывод команды `ifconfig` в

файл, а затем разместить этот файл на форуме, где вам помогут разобраться с этой проблемой. А можно перенаправить список всех процессов (командой `ps -ax`) команде `grep`, которая найдет в списке интересующий вас процесс.

Рассмотрим следующую команду:

```
echo "some text" > file.txt
```

Символ > означает, что вывод команды, находящейся слева от этого символа, будет записан в файл, находящийся справа от символа, при этом файл будет перезаписан.

Чуть ранее мы говорили о перенаправлении вывода программы `ifconfig` в файл. Команда будет выглядеть так:

```
ifconfig > ifconfig.txt
```

Если вместо > указано >>, то исходный файл не будет перезаписан, а вывод команды добавится в конец файла:

```
echo "some text" > file.txt
echo "more text" >> file.txt
cat file.txt
some text
more text
```

Кроме символов > и >> для перенаправления ввода/вывода часто употребляется вертикальная черта `|`. Предположим, что мы хотим вывести содержимое файла big_text:

```
cat big_text
```

Но в файле big_text много строк, поэтому мы ничего не успеем прочитать. Следовательно, целесообразно отправить вывод команды `cat` какой-нибудь программе, которая будет выводить файл постранично, например:

```
cat big_text | more
```

Конечно, этот пример не очень убедительный, потому что для постраничного вывода гораздо удобнее команда `less`:

```
less big_text
```

Вот еще один интересный пример. Допустим, мы хотим удалить файл file.txt без запроса — для этого можно указать команду:

```
echo y | rm file.txt
```

Команда `rm` запросит подтверждение удаления (нужно нажать клавишу <Y>), но за нас это сделает команда `echo`.

И еще один пример. Пусть имеется большой файл, и нам нужно найти в нем все строки, содержащие подстроку 555-555. Чтобы не делать это вручную, можно воспользоваться командой:

```
cat file.txt | grep "555-555"
```

В Linux принято сообщения об ошибках выводить не просто на стандартный вывод (т. е. экран), а на стандартный поток ошибок. По умолчанию, когда не используется перенаправление ввода/вывода, ошибки выводятся на экран. Но вы можете перенаправить сообщения об ошибках в другой файл:

```
gcc 2>gcc.errors
```

Ошибка при запуске `gcc` будет перенаправлена в файл gcc.errors. Какая может быть ошибка? Так ведь мы не указали входящие файлы! Поэтому в файл gcc.errors будет записано сообщение:

```
gcc: no input files
```

8.2. Библиотечные функции C для организации ввода/вывода

К механизмам библиотечного ввода/вывода можно отнести следующие механизмы: функции открытия и закрытия файлов, функции чтения и записи файлов, а также механизмы произвольного доступа к файлам.

В стандартной библиотеке C объявлен тип данных `FILE` — это своеобразная абстракция файла:

```
FILE * file1;
```

Стандартные ввод, вывод и поток ошибок представлены глобальными переменными:

```
extern FILE * stdout;        /* Стандартный вывод */
extern FILE * stdin;  /* Стандартный ввод */
extern FILE * stderr; /* Стандартный поток ошибок */
```

В предыдущем разделе было сказано, что мы ограничимся только файловым вводом/выводом. Весь ввод/вывод в Linux сводится к файловому вводу/выводу. Все, что может принимать и отправлять данные (устройства, сетевые соединения и т. д.), представлено в виде файлов.

Рассмотрим стандартные библиотечные функции открытия и закрытия файлов:

```
FILE * fopen (const char * FLOCATION, const char * OPEN_MODE);
FILE * freopen (const char * FLOCATION, const char * OPEN_MODE, FILE * FP);
FILE * fclose (FILE *FP);
```

Первая функция открывает файл, имя файла указано первым параметром, режим доступа — вторым. Функция `fopen()` возвращает указатель на поток, за которым закреплен файл, указанный в первом параметре. Вторая функция связывает существующий поток `FP` с другим файлом, который задан первым параметром, второй параметр — режим доступа к файлу. Третья функция закрывает поток `FP`.

Посимвольное чтение и запись реализованы в стандартной библиотеке следующими функциями:

```
int fgetc (FILE * FP);
int fputc (int byte, FILE * FP);
```

Первая функция читает байт из потока `FP`, вторая функция записывает байт `byte` в поток `FP`.

Понятно, что посимвольно записывать и читать данные не всегда удобно, для чтения и записи строк используются следующие функции:

```
char * fgets (char * STR, int SIZE, FILE * FP);
int fputs (const char * STR, FILE * FP);
int fprintf(FILE * FP, const char * FMT, ...);
int fscanf (FILE * fp, const char * FMT, ...);
int vfscanf (FILE * FP, const char * FMT, va_list ap);
int vfprintf (FILE * FP, const char * FMT, va_list ap);
```

Думаю, вы знакомы с этими функциями. Функции `fgets()` и `fputs()` соответственно читают строку из потока `FP` и записывают строку в поток. Параметр `SIZE` функции `fgets()` задает размер строки `STR`.

Функции `fprintf()` и `fscanf()` аналогичны функциям `printf()` и `scanfs()`, но работают не со стандартным вводом/выводом, а с файлами. Функции `vfscanf()` и `vfsprintf()` впервые появились в спецификации C99. Они подобны функциям `fprintf()` и `fscanf()`, но список аргументов заменен указателем на этот список. Указатель должен иметь тип `va_list`, который определен в заголовке stdarg.h.

Обычно файловый ввод/вывод осуществляется последовательно, поэтому к абстракции привязывается понятие текущей позиции ввода/вывода. Данная позиция устанавливается в начало файла в момент его открытия. Каждая операция чтения/записи перемещает эту позицию вперед, в соответствии с количеством прочитанных/записанных файлов. Используя механизмы произвольного доступа, вы можете принудительно изменять текущую позицию.

В C произвольный доступ к данным осуществляется с помощью следующих механизмов:

```
#define SEEK_SET 0          /* Начало файла */
#define SEEK_CUR 1          /* Текущая позиция */
#define SEEK_END 2          /* Конец файла */
typedef struct {...} fpos_t;
int fseek (FILE * FP, long int OFFSET, int ORIGIN);
int fgetpos (FILE * FP, fpos_t * POSITION);
int fsetpos (FILE * FP, fpos_t * POSITION);
long int ftell (FILE * FP);
void rewind (FILE * FP);
```

Функция `fseek()` устанавливает указатель текущей позиции в файле. Первый параметр — указатель файла, второй — смещение (указывается в количестве байтов) от начала отсчета (`ORIGIN`). Третий параметр (начало отсчета) может быть одним из следующих макросов: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`.

Функция `fgetpos()` возвращает текущую позицию файла:

```
FILE *fp;
fpos_t f_location;
...
fgetpos(fp, &f_location);
```

Функция `fsetpos()` устанавливает позицию файла. Позиция файла может быть получена с помощью `fgetpos()`:

```
fsetpos(fp, &f_location);
```

Функция `ftell()` возвращает текущее значение указателя позиции файла для потока `FP`, в случае с двоичными файлами возвращаемое значение равно количеству байтов, отделяющих указатель от начала файла. В случае ошибки `ftell()` возвращает –1.

Последняя функция, `rewind()`, перемещает указатель текущей позиции файла в начало потока. Все эти функции представлены в заголовочном файле stdio.h.

При написании программ, использующих стандартные функции C для файлового ввода/вывода, можно также использовать очень полезную функцию `fflush()`, сбрасывающую содержимое буферов ввода/вывода на диск. При вызове `fflush()` производится физическая запись в файл, но файл при этом не закрывается. Вызов `fflush()` является гарантией того, что данные будут записаны на диск. Функции `fflush()` нужно передать только указатель файла:

```
int fflush(FILE * stream);
```

Рассмотрим несколько примеров использования стандартной библиотеки C. Напишем простую программу, которая читает и выводит на стандартный вывод содержимое текстового файла (листинг 8.1).

Листинг 8.1. Чтение текстового файла

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char * argv[])
{
FILE *fp;
char c;

       if ((fp=fopen(argv[1], "r"))==NULL) {
              printf("Ошибка при открытии файла\n");
              exit(1);
       }

       while ((ch=fgets(fp)) != EOF) {
              printf("%c", ch);
       }
```

```
        fclose(fp);

        return 0;
}
```

Данная программа открывает файл, указанный ей в качестве первого параметра, читает его посимвольно и выводит на экран. Мы использовали только библиотечные функции C. Теперь напишем еще одну программу, опять-таки использующую только библиотечные функции. На этот раз программа будет копировать один файл в другой (листинг 8.2).

Листинг 8.2. Программа stdcopy.c

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[] )
{
FILE *in, *out;
char *ch;

if (argc!=3) {
        printf("Использование: stdcopy in-file out-file\n");
        exit(1);
}

if(in=fopen(argv[1], "rb"))==NULL) {
        printf("Ошибка при открытии исходного файла \n");
        exit(1);
}

if(out=fopen(argv[2], "wb"))==NULL) {
        printf("Ошибка при открытии результирующего файла \n");
        exit(1);
}

while(!feof(in)) {
        ch = fgetc(in);
        if(ferror(in)) { printf("Ошибка чтения\n"); break; }
        else {
                if(!feof(in)) putc(ch, out);
                if(ferror(out)) { printf("Ошибка записи\n"); break; }
        }
}

fclose(in);
fclose(out);
return 0;
}
```

Алгоритм программы предельно прост: сначала открываются оба файла. Затем программа посимвольно читает исходный файл и записывает прочитанные байты в результирующий файл.

8.3. Низкоуровневый ввод/вывод

8.3.1. Системные вызовы файлового ввода/вывода

Попытаемся сравнить библиотечные функции языка C с низкоуровневым вводом/выводом. Начнем с абстракции файла. В Linux вместо абстракции файла (указателей типа `FILE`) используются файловые дескрипторы. Файловый дескриптор — это только звучит грозно, а на самом деле — это просто целое число.

Для каждого процесса ядро создает персональную таблицу файловых дескрипторов, в которых и хранится информация об открытых файлах. Ядро ограничивает размер таблицы файловых дескрипторов. Узнать текущее ограничение можно командой `ulimit -n`, которая присутствует в оболочках `bash`, `ksh` и некоторых других. В `csh` и `tcsh` для этого используется команда `limit descriptors`.

Для открытия и закрытия файлов предназначены системные вызовы `open()`, `creat()` и `close()`:

```
int open (const char * FNAME, int OFLAGS, mode_t MODE);
int open (const char * FNAME, int OFLAGS);
int creat (const char * FNAME, mode_t MODE);
int close (int FD);
```

Для чтения и записи файлов используются следующие системные вызовы:

```
ssize_t read (int FD, void * BUFFER, size_t SIZE);
ssize_t write (int FD, const void * BUFFER, size_t SIZE);
ssize_t readv (int FD, const struct iovec * VECTOR, int SIZE);
ssize_t writev (int FD, const struct iovec * VECTOR, int SIZE);
```

Для обеспечения произвольного доступа к файлу используется системный вызов `lseek()`:

```
off_t lseek (int FD, off_t OFFSET, int WHENCE);
```

Еще раз отмечу, что все эти функции реализованы в ядре Linux. Теперь рассмотрим эти системные вызовы подробнее. Но сначала нужно поговорить о типах файлов, с которыми вы можете столкнуться в Linux (табл. 8.1). Также в табл. 8.1 приведены константы описания типа файла, которые вы будете использовать при создании программ.

У использования библиотечных функций C есть одно неоспоримое преимущество — простота. Вы можете программировать и не задумываться об особенностях файловой системы Linux, но в этом случае вы не сможете полностью научиться программировать именно для Linux. При программировании в Linux вы должны учитывать режим файла, объединяющий в себе тип файла и права доступа к нему.

Таблица 8.1. *Типы файлов в Linux*

Тип файла	Константа	Описание
Обычный файл	S_IFREG	Самый обычный файл, предназначен для хранения данных. К этому типу относятся не только текстовые файлы, но и двоичные файлы (архивы, исполняемые файлы, библиотеки)
Блочное устройство	S_IFBLK	Все устройства в Linux представлены в виде файлов (об этом мы поговорим в *главе 14*). С блочными устройствами обмен информацией осуществляется сразу блоками данных, в отличие от символьных устройств. Пример блочного устройства — жесткий диск
Символьное устройство	S_IFCHR	Обмен информацией с этим устройством осуществляется посимвольно. Пример символьного устройства — модем, терминал
Каталог	S_IFDIR	Каталог — это тоже файл, он хранит информацию о файлах, которые в нем находятся. Операции над каталогами будут рассмотрены в *главе 15*
Символическая ссылка	S_IFLNK	Указывает на другой файл
Сокет	S_IFSOCK	Используются для организации межпроцессного взаимодействия. Универсальность сокетов заключается в том, что с их помощью можно организовать взаимодействие систем разного типа: Web-сервер обычно работает под управлением Linux или FreeBSD, а компьютер пользователя может работать под управлением Windows или MacOS
Канал FIFO	S_IFIFO	Еще одно средство организации межпроцессного взаимодействия. Подробно сокеты и каналы FIFO будут рассмотрены в *главе 12*

При написании программ с применением обычных библиотечных функций C вы использовали режим доступа к файлу — вам же нужно указать, для чего вы открываете файл (для чтения или для записи), а вот режим файла — это особенность файловой системы Linux.

Режим файла в Linux представляет собой цепочку из двух байтов (16 битов), которые можно разделить на три группы (табл. 8.2). Часто режим файла (file mode) путают с правами доступа (permissions), но это не так: права доступа являются частью режима файла.

Таблица 8.2. *Режим файла в Linux*

Биты	Описание
0–8	Стандартные права доступа
9–11	Расширенные права доступа
12–15	Тип файла

Поговорим о стандартных правах доступа. Бывшему Windows-пользователю сначала будет трудно привыкнуть к правам доступа в Linux, поэтому им в этой книге посвящена вся *глава 18*. Сейчас мы вкратце рассмотрим концепцию прав доступа, чтобы вы могли уже сейчас приступить к написанию программ.

Получить доступ к файлу могут: владелец файла (пользователь, создавший файл), группа владельца (пользователи, состоящие в одной с владельцем группе пользователей) и все остальные пользователи. Суперпользователь root стоит выше этой концепции и может получить доступ к любому файлу любого пользователя, независимо от установки прав доступа.

Для каждой категории (владелец, группа и все остальные) устанавливаются отдельные права доступа к файлу: право чтения, право записи и право выполнения. Право чтения означает, что файл можно читать. Право записи означает, что файл может быть изменен.

Последнее право, право выполнения, устанавливается только для программ. Пока вы не установите это право, система не будет пытаться выполнить исполнимый файл. Право выполнения для каталогов означает, что пользователи могут просматривать содержимое каталога.

При написании программ стандартные права доступа указываются с помощью констант режима, описанных в заголовочном файле sys/stat.h. Данные константы описаны в табл. 8.3.

Таблица 8.3. *Константы стандартных прав доступа*

Константа	Бит	Описание
S_IXOTH	0	Право выполнения для остальных пользователей
S_IWOTH	1	Право записи для остальных пользователей
S_IROTH	2	Право чтения для остальных пользователей
S_IXGRP	3	Право выполнения для группы владельца
S_IWGRP	4	Право выполнения для группы владельца
S_IRGRP	5	Разрешает чтение файла для группы владельца
S_IXUSR	6	Право выполнения для владельца
S_IWUSR	7	Право записи для владельца
S_IRUSR	8	Разрешает чтение файла владельцем

Константу вы будете использовать при написании программ, а колонка "Бит" приведена в табл. 8.3 для большего понимания структуры режима файла. В табл. 8.4 указаны расширенные права доступа (подробно они рассмотрены в *главе 18*, а пока лишь ознакомьтесь с константами).

В табл. 8.5 приведены самые используемые флаги открытия файлов. Эти флаги похожи на режим доступа файла в стандартных библиотечных функциях C, но указываются немного на другой лад.

Таблица 8.4. *Константы для расширенных прав доступа*

Константа	Описание
S_ISUID	Бит SUID
S_ISGID	Бит SGID
S_ISVTX	"Липкий" бит

Таблица 8.5. *Флаги открытия файла*

Константа	Описание
O_RDONLY	Открыть файл только для чтения
O_WRONLY	Открыть файл только для записи
O_RDWR	Файл открывается и для чтения, и для записи
O_APPEND	Дозапись в конец файла
O_ASYNC	Будет генерироваться сигнал SIGIO, если ввод/вывод для данного файла возможен
O_CREAT	Создать файл, если он не существует
O_DIRECT	Кэширование файла будет производиться по минимуму, подходит для важных файлов, когда нужно сразу физически записывать данные на диск
O_DIRECTORY	Ошибка, если файл не является каталогом
O_EXCL	Генерирует ошибку, если файл существует
O_NOFOLLOW	Генерирует ошибку, если файл не является ссылкой
O_NONBLOCK	Открыть файл в неблокируемом режиме, если это возможно
O_NOATIME	Не обновлять время доступа к файлу
O_SYNC	Файл открывается для синхронного ввода/вывода: каждая операция write() ожидает физической записи данных на диск
O_TRUNC	Устанавливает длину файла в 0, если он существует, является обычным файлом и есть возможность записи

Как видите, констант очень много. Указать несколько констант можно так:

```
S_IXUSR | S_IWUSR
```

В данном случае предоставляются права чтения и записи для владельца.

8.3.2. Системный вызов *creat()*

Вот теперь можно рассмотреть системный вызов creat(), описанный в файле fcntl.h:

```
int creat (const char * FILENAME, mode_t MODE);
```

Первый параметр задает имя файла, второй — режим файла. Значение –1 возвращается в случае ошибки: файл не получилось открыть или создать. Теперь напишем программу, использующую системный вызов `creat()` (листинг 8.3).

Листинг 8.3. Программа создания файла

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main (void)
{
        int fdesc;
        mode_t modes = S_IRUSR | S_IWUSR;

        fdesc = creat("test.txt", modes);
        if (fdesc == -1) {
                fprintf(stderr, "Ошибка при создании файла\n");
                return 1;
        }

        close(fdesc);
        return 0;
}
```

Наша простейшая программа пытается создать файл test.txt: в случае ошибки на стандартный поток ошибок будет выведено соответствующее сообщение. Обратите внимание, как происходит вывод на поток ошибок: мы просто записываем строку в файл stderr.

Даже в случае успешного создания файла мы просто закрываем его с помощью системного вызова `close()`, который описан в заголовочном файле unistd.h. В этом заголовочном файле описаны также системные вызовы `write()`, `read()`, `lseek()` и константы `SEEK_CUR`, `SEEK_SET` и `SEEK_END`.

Заголовочный файл fcntl.h мы подключили потому, что в нем описан системный вызов `open()` и константы `O_RDONLY`, `O_WRONLY` и др. Константы режима файла (`S_IRUSR`, `S_IWUSR`) описаны в файле sys/stat.h. В файле sys/types.h описаны важные системные типы.

8.3.3. Чтение файла: системные вызовы *open()* и *read()*

В стандартной библиотеке C функция `fopen()` используется как для создания, так и для открытия файла. При низкоуровневом вводе/выводе используется немного иная концепция: для создания файла используется один системный вызов (`creat`), а для открытия — другой (`open`).

Это связано с тем, что при создании файла задаются, как правило, права доступа и другие параметры, характерные операции создания. При открытии файла вы не можете изменить права доступа, но можете указать режим доступа — как вы хотите открыть файл — для чтения, записи или дозаписи в конец файла и т. д. Флаги открытия файла были описаны в табл. 8.5.

Прототипы системного вызова `open()` представлены далее:

```
int open (const char * FILENAME, int FLAGS, mode_t MODE);
int open (const char * FILENAME, int FLAGS);
```

Первый параметр — имя файла, второй — флаги открытия файла, третий — режим файла. Режим файла можно не указывать и ограничиться двумя параметрами.

Системный вызов `read()` объявлен в заголовочном файле unistd.h, его прототип выглядит так:

```
ssize_t read (int FD, void * BUFFER, size_t SIZE);
```

Первый параметр — это дескриптор файла, второй — буфер, в который будут записаны прочитанные из файла данные (в качестве буфера обычно используют строки). `SIZE` — размер (в байтах) данных, которые нужно прочитать.

Системный вызов `read()` возвращает:

- количество реально прочитанных байтов — количество прочитанных байтов может отличаться от параметра `SIZE`: например, вы хотите прочитать 100 байтов, а в файле физически осталось пять байтов;
- 0 — когда достигнут конец файла;
- –1 — произошла ошибка при чтении файла.

Напишем программу, демонстрирующую посимвольное чтение файла (листинг 8.4).

Листинг 8.4. Посимвольное чтение файла с использованием системных вызовов

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main (void)
{
        int fd;
        char c;

        fd = open("test.txt", O_RDONLY);

        if (fd == -1)
        {
                fprintf(stderr, "Не могу открыть файл\n");
                return 1;
        }
```

```
        while (read (fd, &c, 1) > 0) printf("%c", c);
        close(fd);
        return 0;
}
```

Нужно отметить, что приведенную нами программу лучше не использовать для чтения больших файлов, т. к. она крайне неэффективна. Наша программа станет работать намного быстрее, если читать файл большими блоками, например по 4096 байт. Если же читать посимвольно, то для чтения простого файла в 4 Кбайт наша программа произведет 4096 обращений к ядру!

8.3.4. Системный вызов *write()*

Системный вызов `write()` используется для записи информации в файл, функция `write()` объявлена в заголовочном файле unistd.h:

```
ssize_t write (int FD, const void * BUFFER, ssize_t SIZE);
```

Запись производится в файл, заданный файловым дескриптором `FD`, чтение производится из буфера `BUFFER`, записывается `SIZE` байтов. Функция возвращает количество реально записанных байтов или –1 в случае ошибки.

Чуть раньше мы рассмотрели программу stdcopy.c, выполняющую копирование файлов с использованием стандартных функций C. Теперь мы напишем аналогичную функцию (syscopy.c), выполняющую те же действия, но с использованием системных вызовов Linux (листинг 8.5).

Листинг 8.5. Программа syscopy.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

int main (int argc, char ** argv)
{
        /* Буфер размером 4096 байтов */
        char buffer [4096];

        /* Дескрипторы исходного и результирующего файла */
        int fin, fout;

        ssize_t r;      /* К-во прочитанных байтов */

        /* Проверяем количество аргументов */
        if (argc < 3) {
                fprintf(stderr, "Использование: syscopy in out\n);
                return 1;
        }
```

```
        /* Открываем исходный файл */
        fin = open (argv[1], O_RDONLY);
        if (fin == -1) {
        /* В этот раз выводим сообщение об ошибке просто на
        стандартный вывод */
               printf("Не могу открыть файл: %s\n", argv[1]);
               return 1;
        }

        /* Открываем результирующий файл */
        fout = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0640);
        if (fout == -1)
        {
        printf("Не могу открыть файл: %s\n", argv[2]);
        return 2;
        }

        /* Копирование файлов */
        while ((r = read(fin, buffer, 4096)) > 0)
               write(fout, buffer, r);

        /* Закрываем файлы */
        close(fin);
        close(fout);

        return 0;
}
```

Программа очень проста: сначала открываются два файла (файлы указываются в командной строке), их дескрипторы: `fin` (входной файл) и `fout` (результирующий файл). В случае ошибки программа выведет соответствующее сообщение.

Обратите внимание, как открывается результирующий файл:

```
fout = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0640);
```

Мы открываем файл для записи (`O_WRONLY`), создаем при необходимости (`O_CREAT`) или же удаляем, если такой файл уже существует (`O_TRUNC`). Права доступа для нового файла устанавливаются 0640: владельцу разрешается чтение и запись файла, группе — только чтение, а остальным пользователям запрещается любой доступ к файлу. Подробно о правах доступа мы поговорим в *главе 18*.

Далее в цикле `while` мы читаем входной файл блоками по 4 Кбайт и записываем прочитанные блоки в результирующий файл. Мы не обрабатываем результат функции `write()` для упрощения кода программ. Затем мы закрываем оба файла.

Перед тем как приступить к рассмотрению следующего системного вызова, хочется сказать еще несколько слов о системном вызове `write()`. Первый параметр у данного системного вызова — дескриптор файла. Вместо дескриптора файла вы можете

указать stdout (значение 1) или stderr (значение 2). Таким образом мы можем вывести сообщение на стандартный вывод или на стандартный поток ошибок:

```
write(2, "Error", 5);
```

8.3.5. Системный вызов *lseek()*

С помощью системного вызова lseek() мы можем произвольно перемещать текущую позицию ввода/вывода. Системный вызов объявлен в заголовочном файле unistd.h так:

```
off_t lseek(int FD, off_t OFFSET, int ORIGIN);
```

Первый параметр — это дескриптор файла, второй — смещение в байтах от начальной позиции, заданной аргументом ORIGIN. В качестве ORIGIN можно использовать константы:

- SEEK_SET — начало файла;
- SEEK_CUR — текущая позиция ввода/вывода;
- SEEK_END — конец файла.

В случае ошибки lseek() возвращает –1, а если вызов удался, то возвращается установленная позиция ввода/вывода относительно начала файла.

В Linux нет системных вызовов, которые бы "перематывали" файл в самое начало или возвращали бы текущую позицию, но их можно реализовать с помощью lseek() так:

```
/* "перемотка" в начало файла */
lseek(fd, 0, SEEK_SET);
/* Получаем текущую позицию */
off_t position = lseek(fd, 0, SEEK_CUR);
```

Самое время перейти к следующей части книги, посвященной системному программированию.

ЧАСТЬ III

Системное программирование

Глава 9.	Концепция многозадачности
Глава 10.	Системные вызовы для работы с процессами
Глава 11.	Многопоточные приложения
Глава 12.	Взаимодействие процессов
Глава 13.	Создание модуля ядра

Третья часть книги посвящена системному программированию. При разработке обычных пользовательских приложений материалы этой части, может, и не важны, однако это не означает, что можно сразу же перейти к следующей части книги. Как минимум, системное программирование — это интересно, ведь вы знакомитесь с "недрами" операционной системы и начинаете понимать, как она устроена.

ГЛАВА 9

Концепция многозадачности

9.1. Основы многозадачности Linux

9.1.1. Иерархия процессов

Как мы уже знаем, Linux — многозадачная система. Многозадачность построена на иерархии процессов. Как уже было отмечено, ничего так просто не запускается — всегда находится процесс, который запустит следующий процесс. На вершине иерархии — программа `init` (да, она играет роль системы инициализации Linux во многих дистрибутивах).

Когда вы введете команду `mc`, запустится файловый менеджер Midnight Commander, который будет *дочерним процессом* (или *потомком* — так проще) по отношению к `bash`. В свою очередь, `bash` будет *родительским процессом* (или *родителем*).

В Linux программа `init` — единственный процесс без родителя (программа `init` запускается непосредственно ядром при запуске системы). Кстати, у ядра есть параметр `init`, с помощью которого можно изменить систему инициализации — тогда вместо `init` будет запущена другая программа.

Каждому процессу в Linux присваивается уникальный номер — идентификатор процесса (PID, Process ID). Идентификатор процесса `init` равен 1.

Зная ID процесса, вы можете управлять процессом, а именно завершить процесс или изменить приоритет процесса. Принудительное завершение процесса необходимо, если процесс завис и его нельзя завершить обычным образом. А изменение приоритета может понадобиться, если вы хотите, чтобы процесс доделал свою работу быстрее.

Для получения информации о процессах используется команда `ps`. Чтобы увидеть дерево процессов, введите команду `ps -e --forest`. Вывод программы `ps` не очень удобен (рис. 9.1). Возможно, в вашем дистрибутиве установлены другие инструменты для вывода дерева процессов, например программа `pstree`, вывод которой более удобен (рис. 9.2). Учитывая, что дерево процессов довольно большое, я бы рекомендовал выводить постранично:

```
pstree | less
```

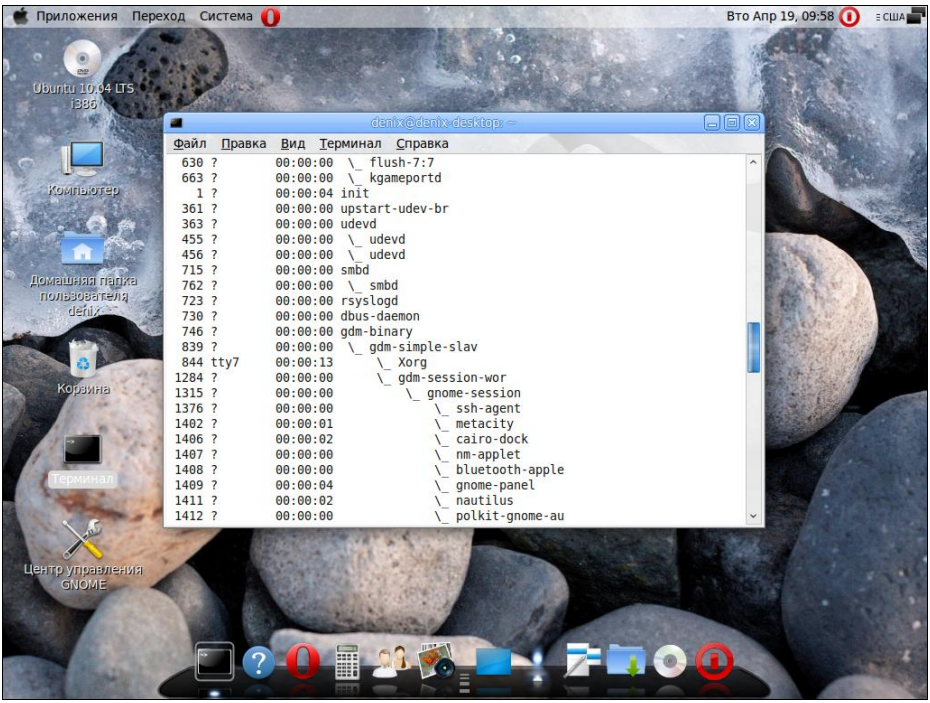


Рис. 9.1. Команда `ps -e --forest`

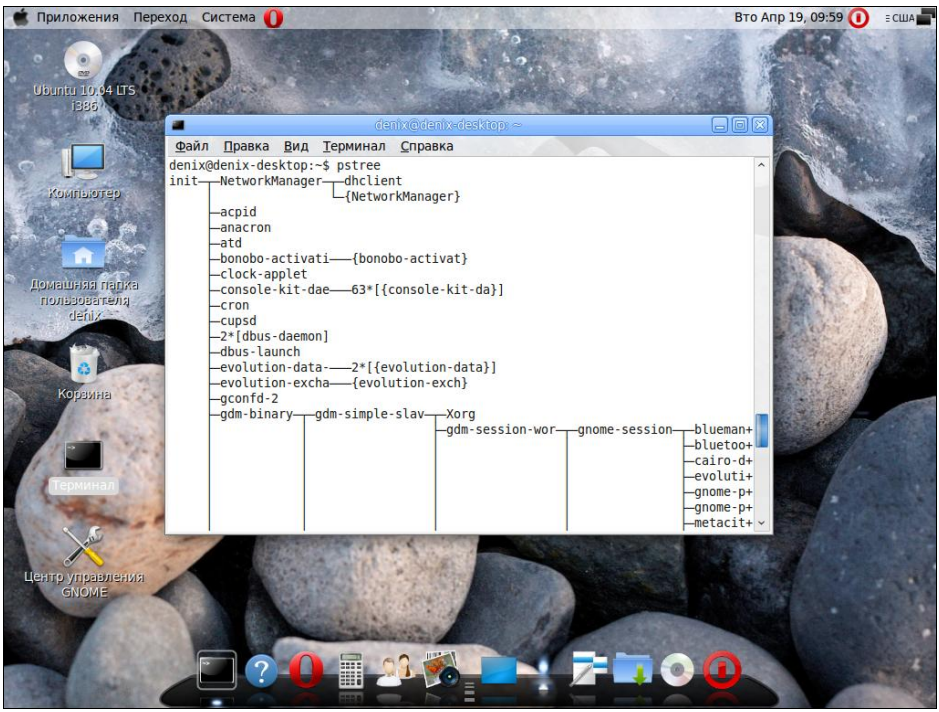


Рис. 9.2. Команда `pstree`

Часто используемые параметры программы ps представлены в табл. 9.1.

Таблица 9.1. *Параметры программы ps*

Параметр	Описание
-a	Отобразить все процессы, связанные с терминалом (отображаются процессы всех пользователей)
-e	Отобразить все процессы
-t список_терминалов	Отобразить процессы, связанные с указанными терминалами
-u идентификаторы_пользователей	Отобразить процессы, связанные с данными идентификаторами
-g идентификаторы_групп	Отобразить процессы, связанные с данными идентификаторами групп
-x	Отобразить все процессы, не связанные с терминалом

9.1.2. Аварийное завершение процесса

Предположим, у вас зависла какая-то программа, например файловый менеджер mc. Хоть это и маловероятно (не помню, чтобы он когда-нибудь зависал), но для примера пусть будет так. Принудительно завершить ("убить") процесс можно с помощью команды kill. Формат ее вызова следующий:

```
kill [параметры] PID
```

PID (Process ID) — это идентификатор процесса, который присваивается процессу системой; уникален для каждого процесса. Но мы знаем только имя процесса (имя команды), но не знаем идентификатор процесса. Узнать идентификатор процесса позволяет программа ps. Предположим, что mc находится на первой консоли. Поскольку он завис, вы не можете более использовать консоль, и вам нужно переключиться на вторую консоль (клавиатурной комбинацией <Alt>+<F2>). Зарегистрировавшись на второй консоли, введите команду ps. Она выведет список процессов, запущенных на второй консоли, — это будет bash и сам ps (рис. 9.3).

Чтобы добраться до нужного нам процесса (mc), который запущен на первой консоли, введите команду ps -a или ps -U root. В первом случае вы получите список процессов, запущенных вами, а во втором — список процессов, запущенных от вашего имени (я предполагаю, что вы работаете под именем root).

Обратите внимание — вы сами запустили процессы mc и ps (рис. 9.4), а от вашего имени (root) система запустила множество процессов. Следует заметить, что программа ps выводит также имя терминала (tty1), на котором запущен процесс. Это очень важно — если на разных консолях у вас запущены одинаковые процессы, можно легко ошибиться и завершить не тот процесс. Для графических процессов номер терминала не выводится — вместо него в колонке TTY вы увидите вопросительный знак (?). Если процесс запущен на псевдотерминале (например, вы работаете в

графическом режиме, открыли эмулятор терминала и запустили в нем процесс), то `ps` отобразит номер псевдотерминала так: `pts/N`, где `N` — число.

Теперь, когда мы знаем PID нашего процесса, мы можем его "убить":

```
# kill 2484
```

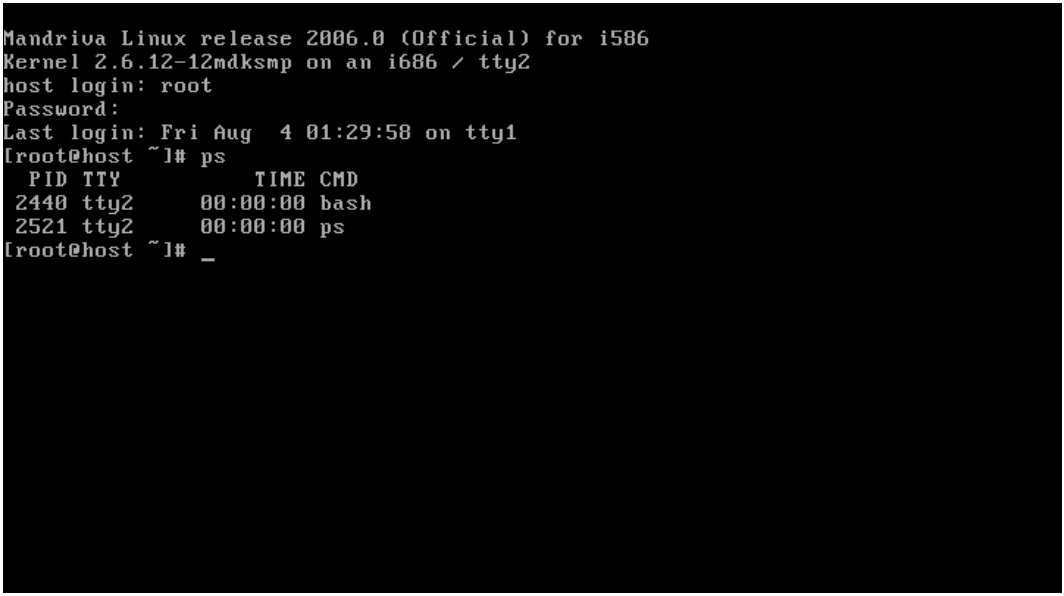


Рис. 9.3. Список процессов на текущей консоли

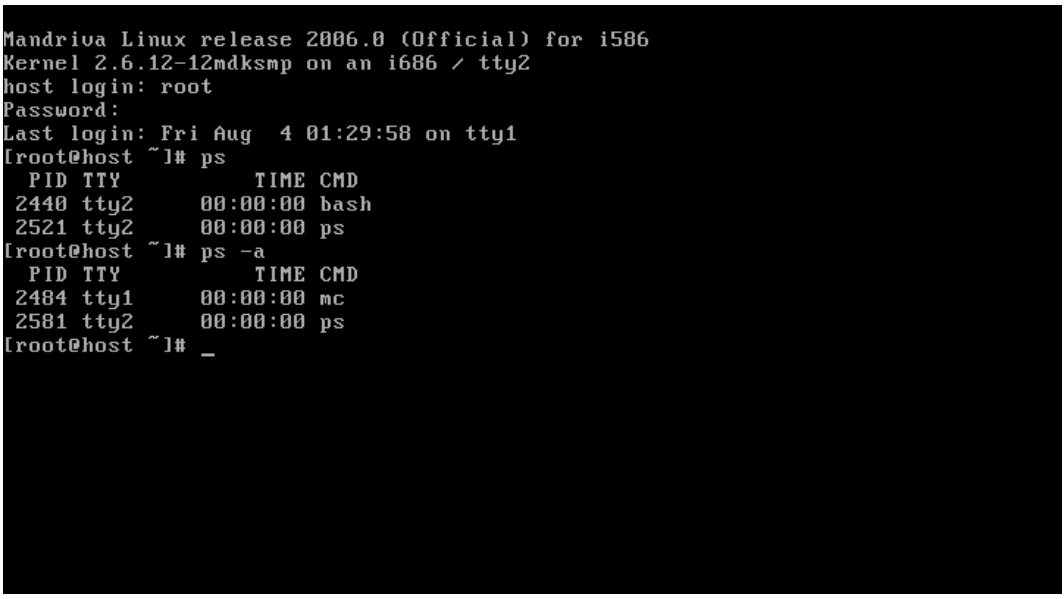


Рис. 9.4. Определение PID программы `mc`

Перейдите на первую консоль после выполнения этой команды — mc на ней уже не будет. Если выполнить команду ps -a, то в списке процессов mc тоже не будет.

Проще всего вычислить PID процесса с помощью следующей команды:

```
# ps –ax | grep <имя>
```

Например, # ps –ax | grep firefox.

Вообще-то, все эти действия, связанные с вычислением PID процесса, мы рассмотрели только для того, чтобы познакомиться с командой ps.

Так что, если вы знаете только имя процесса, гораздо удобнее использовать команду:

```
# killall <имя процесса>
```

Но имейте в виду, что данная команда завершит все экземпляры данного процесса. А вполне может быть, что у нас на одной консоли находится mc, который нужно "убить", а на другой — нормально работающий mc. Команда killall "убьет" оба процесса.

При выполнении команд kill и killall нужно помнить, что если вы работаете от имени обычного пользователя, они могут завершить только те процессы, которые принадлежат вам. А если вы работаете от имени пользователя root, то можете завершить любой процесс в системе.

Кроме команды kill пользователи, предпочитающие графический интерфейс, могут использовать программу xkill, позволяющую "убить" графическую программу. Введите команду xkill, указатель мыши изменится на череп, которым нужно будет щелкнуть по зависшему окну (иначе, если окно не зависло, зачем его "убивать"?). Процесс, относящийся к выбранному окну, будет немедленно завершен.

После того как вы узнали о существовании команды kill, наверное, вам интересно будет знать, что случится с дочерним процессом, если родительский процесс прекратит свою работу (или самостоятельно, или вы ему поможете командой kill). Давайте посмотрим, что же случится. Введите команды:

```
bash
sleep 1m &
ps -f
```

Первая команда запускает еще один экземпляр командного интерпретатора bash. Вторая команда ничего не делает — она переходит в фоновый режим (благодаря символу &) и просто ждет 1 минуту (нам этого вполне хватит для демонстрации). Третья команда выводит список процессов. Посмотрите на колонку PPID (Parent PID) — это идентификатор родительского процесса. Для нашего процесса sleep идентификатор родительского процесса равен 2048.

Затем введите команду exit, чем вы завершите запущенный интерпретатор bash, тем самым вы завершаете родительский процесс для процесса sleep. Опять введите команду ps -f и посмотрите на идентификатор процесса sleep: он равен 1. Другими словами, родительским процессом для нашего "осиротевшего" процесса sleep стал процесс init (рис. 9.5).

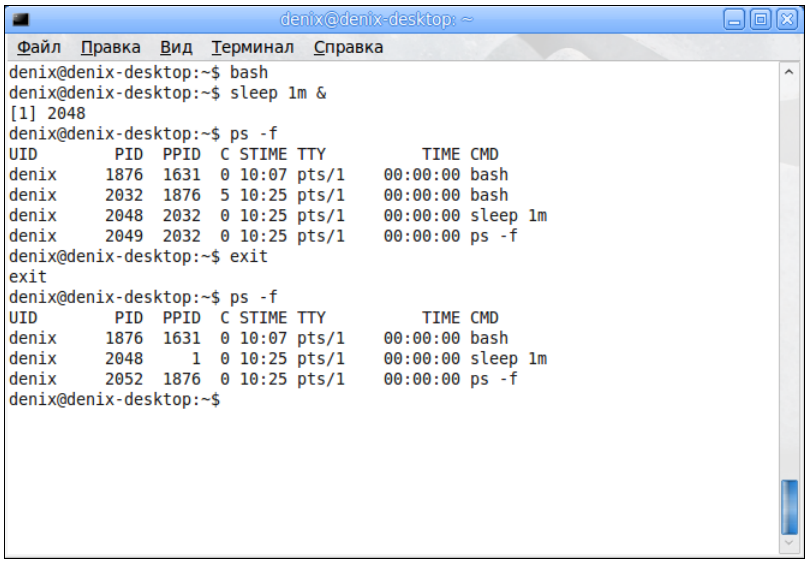


Рис. 9.5. В поисках нового родительского процесса

Однако такое случается не всегда. В некоторых случаях (когда происходит потеря управляющего термина и в некоторых других) дочерний процесс также будет завершен.

9.1.3. Программа *top*: кто больше всех расходует процессорное время

Иногда бывает, что система ужасно тормозит — весь день работала нормально, и вдруг начала притормаживать.

Если вы даже не догадываетесь, из-за чего это случилось, вам нужно использовать программу `top` (рис. 9.6) — она выводит список процессов с сортировкой по процессорному времени. То есть на вершине списка будет процесс, который занимает больше процессорного времени, чем сама система. Вероятно, из-за него и происходит эффект "торможения".

На рис. 9.6 показано, что больше всего процессорного времени (0,3%) занимает программа `top`. Конечно, в реальных условиях все будет иначе. Выйти из программы `top` можно, нажав клавишу <Q>. Кроме клавиши <Q> действуют следующие клавиши:

- <U> — показывает только пользовательские процессы (т. е. те процессы, которые запустил пользователь, под именем которого вы работаете в системе);
- <D> — изменяет интервал обновления;
- <F> — изменяет столбец, по которому сортируются задачи. По умолчанию задачи сортируются по столбцу %CPU, т. е. по процессорному времени, занимаемому процессом;
- <H> — получить справку по остальным командам программы `top`.

```
top - 01:39:31 up 10 min,  3 users,  load average: 0.00, 0.00, 0.00
Tasks:  58 total,   1 running,  57 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0% us,  0.3% sy,  0.0% ni, 99.7% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    189720k total,    68224k used,   121496k free,     5088k buffers
Swap:   128984k total,        0k used,   128984k free,    38072k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2599 root      16   0  1996 1012  804 R  0.3  0.5   0:00.06 top
    1 root      16   0  1564  540  472 S  0.0  0.3   0:00.55 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      10  -5     0    0    0 S  0.0  0.0   0:00.02 events/0
    5 root      16  -5     0    0    0 S  0.0  0.0   0:00.08 khelper
    6 root      11  -5     0    0    0 S  0.0  0.0   0:00.00 kthread
    8 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 kacpid
   61 root      10  -5     0    0    0 S  0.0  0.0   0:00.03 kblockd/0
   93 root      20   0     0    0    0 S  0.0  0.0   0:00.00 pdflush
   94 root      15   0     0    0    0 S  0.0  0.0   0:00.05 pdflush
   96 root      16  -5     0    0    0 S  0.0  0.0   0:00.00 aio/0
   95 root      25   0     0    0    0 S  0.0  0.0   0:00.00 kswapd0
  684 root      16   0     0    0    0 S  0.0  0.0   0:00.00 kseriod
  766 root      13  -5     0    0    0 S  0.0  0.0   0:00.00 ata/0
  775 root      18   0     0    0    0 S  0.0  0.0   0:00.00 scsi_eh_0
  784 root      16   0     0    0    0 S  0.0  0.0   0:00.02 kjournald
  924 root      15  -4  1564  496  420 S  0.0  0.3   0:00.08 udevd
```

Рис. 9.6. Программа `top`

Назначение столбцов программы `top` указано в табл. 9.2.

***Таблица 9.2.** Назначение столбцов программы `top`*

Столбец	Описание
PID	Идентификатор процесса
USER	Имя пользователя, запустившего процесс
PR	Приоритет процесса
NI	Показатель `nice` (*см. разд. 9.1.4)*
VIRT	Виртуальная память, использованная процессом (в Кбайт)
RES	Размер процесса, не перемещенный в область подкачки (в Кбайт). Этот размер равен размерам сегментов кода и данных, т. е. RES = CODE + DATA
S	Состояние процесса: • R — выполняется; • S — "спит" (режим ожидания), в этом состоянии процесс выгружен из оперативной памяти в область подкачки; • D — "непрерываемый сон" (uninterruptible sleep), из такого состояния процесс может вывести только прямой сигнал от оборудования; • T — процесс в состоянии трассировки или остановлен; • Z (зомби) — специальное состояние процесса, когда сам процесс уже завершен, но его структура еще осталась в памяти
%CPU	Занимаемое процессом процессорное время
%MEM	Использование памяти процессом

Таблица 9.2 *(окончание)*

Столбец	Описание
TIME+	Процессорное время, израсходованное с момента запуска процесса
COMMAND	Команда, которая использовалась для запуска процесса (обычно имя исполнимого файла процесса)

9.1.4. Команды *nice* и *renice*: изменение приоритета процесса

Предположим, что вы работаете с видео и вам нужно перекодировать файл из одного видеоформата в другой. Конвертирование видео занимает много процессорного времени, а хотелось бы все сделать как можно быстрее и уйти раньше домой. Тогда вам поможет программа `nice` — она позволяет запустить любую программу с указанным приоритетом. Ясно, что чем выше приоритет, тем быстрее будет выполняться программа. Формат вызова команды следующий:

```
nice -n <приоритет> команда аргументы
```

Максимальный приоритет задается числом −20, а минимальный числом 19. Приоритет по умолчанию равен 10.

Если процесс уже запущен, то для изменения его приоритета можно использовать команду `renice`:

```
renice -n <приоритет> -p PID
```

9.2. Функция *system()*

Пожалуй, настало время заняться программированием. Самый простой способ создать дочерний процесс — это использовать библиотечную функцию `system()`. Функция `system()`, в отличие от системного вызова `fork()`, не порождает новый процесс, а просто передает переданный ей аргумент командной оболочке /bin/sh.

Функция `system()` объявлена в заголовочном файле stdlib.h. Прототип этой функции следующий:

```
int system(const char * COMMAND);
```

Аргумент `COMMAND` — это команда, которую нужно выполнить. Возвращаемое функцией `system()` значение обычно равно 0, если команда успешно выполнена. Если результат функции `system()` отличен от 0, произошла ошибка.

Давайте рассмотрим небольшой пример использования этой функции (листинг 9.1).

Листинг 9.1. Пример использования функции `system()`

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
        system("sleep 30");
        printf("After sleep\n");
        return 0;
}
```

Скомпилируйте и запустите программу. Функция `system()` ожидает завершения запущенного процесса, поэтому сообщение "`After sleep`" вы увидите через 30 секунд — после завершения запущенной команды.

Если вы не желаете ждать, пока завершится выполнение команды, укажите `&` после самой команды:

```
system("sleep 30 &");
```

В качестве аргумента `COMMAND` можно указать любую команду — как команду оболочки, так и команду, запускающую внешнюю программу:

```
system("mcedit /etc/passwd");
system("cd /");
```

Запуск процессов с помощью функции `system()` не очень хорошая затея, учитывая, что существуют системные вызовы `fork()` и `exec()`, которые будут рассмотрены в следующей главе.

ГЛАВА 10

Системные вызовы для работы с процессами

10.1. Создание и запуск процессов

10.1.1. Модели описания состояний процессов

Как было отмечено в предыдущей главе, использовать функцию `system()` — не очень хорошая идея, хотя в некоторых случаях она действительно бывает полезна. В этой главе мы познакомимся с системным вызовом `fork()` и целым семейством функций `exec()`.

Прежде чем приступить к рассмотрению системного вызова `fork()`, вам нужно познакомиться с двумя моделями управления процессами — трех и пяти состояний процессов.

Термин "процесс" впервые появился очень давно — еще при разработке операционной системы Multics. Процессом считается программа на стадии выполнения. Можно встретить и другие определения процесса. Например, процесс — это выполнение пассивных инструкций программы на процессоре компьютера. Независимо от того, какое определение вы выберете для себя, суть, надеюсь, понятна.

В первых UNIX-системах для описания процессов использовалась модель трех состояний. Эта модель предусматривала три состояния процесса: состояние выполнения, состояние ожидания и состояние готовности. Рассмотрим все эти состояния.

- Выполнение — это активное состояние, во время которого процесс обладает всеми необходимыми ему ресурсами. В этом состоянии процесс непосредственно выполняется процессором.
- Ожидание, в отличие от выполнения, является пассивным состоянием. Процесс не завершен, но и не выполняется. Он заблокирован и чего-то ожидает, например ожидает освобождения какого-нибудь устройства.
- Состояние готовности тоже является пассивным. В этом случае процесс тоже заблокирован, но причина блокировки иная. Если в состоянии ожидания процесс блокируется по собственному желанию — "просит" у системы доступ к устрой-

ству, но ждет, пока устройство освободится. А в состоянии готовности процесс заблокирован по независящим от него причинам.

Представим, что вы написали простейшую программу, выводящую строку "Введите свое имя" и ожидающую ввода от пользователя. Когда пользователь введет свое имя, программа выведет приветствие вида: "Привет, <имя>".

Вы запустили эту программу. Состояние выполнения длится до ожидания ввода от пользователя. Затем наступает состояние ожидания — процесс ждет, пока вы введете свое имя. Он может ждать бесконечно долго — пока вы не нажмете клавишу <Enter> или что-нибудь не случится с самим компьютером. Вы нажимаете клавишу <Enter>. По логике вещей, процесс должен перейти в состояние выполнения, но если процессор в данный квант времени занят выполнением другого процесса, процесс переходит в состояние готовности и ждет, пока процессор займется его выполнением. В нашем простом случае переход от готовности к выполнению занимает настолько мало времени, что вы даже не заметите этого. В этом и вся суть квазипараллельного (почти параллельного) выполнения процессов — когда процессор всего один, а пользователю "кажется", что процессы выполняются параллельно. Если у вас многопроцессорная машина, тогда у вас может одновременно действительно выполняться несколько процессов.

Из состояния готовности процесс может перейти только в состояние выполнения, а вот из состояния выполнения процесс может перейти либо в состояние ожидания, либо в состояние готовности. В состояние готовности процесс может перейти, если квант времени, отведенный на выполнение процессора, вышел. В операционной системе есть специальная программа — планировщик процессов, она и следит за тем, чтобы один процесс не "узурпировал" все процессорное время.

Со временем модель трех состояний процессов усовершенствовалась и превратилась в модель пяти состояний. В модель трех состояний было добавлено два состояния — рождение и смерть процесса. Состояние рождения можно охарактеризовать так: самого процесса еще нет (т. е. процессор еще ничего не выполняет или же занят чем-то другим), но структура процесса уже готова. Понятно, что состояние рождения процесса является пассивным.

Состояние смерти процесса обратно состоянию рождения: процесса уже нет, а структура процесса еще сохранилась. Процессы в состоянии смерти называются *зомби*. Ничего общего с магией Вуду нет: просто система еще не успела стереть структуру процесса из списка процессов. Некоторые процессы могут находиться в состоянии смерти довольно продолжительное время, и мы успеем заметить состояние Z в списке процессов программы `top`. В этой главе мы создадим процесс зомби намеренно.

Модель пяти состояний приведена на рис. 10.1. Из состояния рождения процесс переходит в состояние готовности, т. к. может произойти ситуация, когда во время запуска нашего процесса процессор занят выполнением другого процесса, поэтому наш процессор переходит в состояние готовности. Перейти в состояние смерти можно только из состояния выполнения. В остальном модель пяти состояний полностью повторяет модель трех состояний.

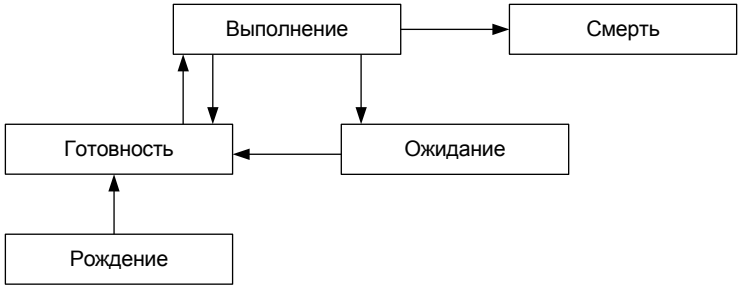


Рис. 10.1. Модель пяти состояний

Рассмотрим некоторые операции над процессами в контексте модели пяти состояний:

- создание процесса — переход из состояния рождения в состояние готовности;
- запуск процесса — переход из состояния готовности в состояние выполнения;
- восстановление процесса — переход из состояния готовности в состояние выполнения;
- блокирование процесса — переход из состояния выполнения в состояние ожидания;
- пробуждение — переход из состояния ожидания в состояние готовности;
- уничтожение процесса — переход из состояния выполнения в состояние смерти.

При создании процесса процессу присваивается имя и идентификатор, добавляется информация в список процессов, определяется приоритет процесса, процессу предоставляются необходимые ему ресурсы.

10.1.2. Особенности *fork()*

В Linux каждый процесс выполняется в собственном виртуальном адресном пространстве, другими словами, процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на всю систему в целом. Один процесс не может прочитать что-либо из памяти другого процесса (или записать в нее) без "разрешения" на то другого процесса. Санкционированные взаимодействия между процессами допускаются системой, мы их рассмотрим в *главе 12*, когда будем изучать способы взаимодействия процессов.

Первые версии Linux не поддерживали SMP (Symmetric Multiprocessor Architectures). С развитием SMP поддержка многопроцессорности появилась и в Linux. Позже в ядро Linux был внедрен механизм потоков (подробно о них мы поговорим в *главе 11*), или нитей. Нить — это процесс, выполняющийся в виртуальной памяти, которая используется вместе с другими нитями одного и того же "тяжеловесного" процесса, который обладает отдельной виртуальной памятью. Нити также называются "легковесными" процессами.

Нити (потоки) позволяют решать в рамках одной программы одновременно несколько задач. Например, вы можете написать программу загрузки файла. Файл

"виртуально" разбивается на несколько частей, каждая нить будет загружать свою часть файла, в результате чего сократится общее время загрузки. Такие программы называются *многопоточными*. Подробно о потоках мы поговорим в *главе 11*.

По сути, каждый поток тоже является как бы процессом. Исходя из всего этого, потоки — это наборы команд, имеющие возможность получать время процессора. Время процессора выделяется квантами. Квант — это минимальное время, на протяжении которого поток (нить) может использовать процессор.

Выполнение программ начинается с так называемой точки входа (entry point). Для типичной С-программы точкой входа является функция `main()`. Наша программа может также создать программу и ее выполнение тоже начнет с функции `main()`.

Представим, что наша программа вызвала системный вызов `fork()`. Этот системный вызов создает новый процесс, при этом будет создано новое адресное пространство, полностью аналогичное адресному пространству основного процесса.

Обратите внимание: системный вызов `fork()` не запускает процесс, а только создает его — он подготавливает фундамент для запуска нового процесса. Однако запуском процесс занимается другой системный вызов — `execl()`.

После выполнения `fork()` вы получите два абсолютно одинаковых процесса — основной и порожденный. После создания нового процесса можно запустить в нем программу с помощью системного вызова `execl()`.

Проведем небольшой эксперимент. Создайте следующее простое приложение (листинг 10.1).

Листинг 10.1. Программа fork-demo.c

```
#include <unistd.h>
#include <stdio.h>

int main (void)
{
        fork();
        sleep(30);
        return 0;
}
```

Скомпилируйте и запустите программу:

```
gcc -o fork-demo fork-demo.c
./form-demo
```

Затем введите команду `ps`. Вы увидите в списке процессов два процесса fork-demo. Это абсолютно два одинаковых процесса, которые будут выполнять одни и те же действия.

Теперь давайте усложним нашу программу: научим ее отличать родительский процесс от дочернего (листинг 10.2). Реализовать это очень просто: для дочернего процесса `fork()` возвращает 0, –1 возвращается в случае ошибки.

Листинг 10.2. Программа fork-demo2.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main (void)
{
       if (fork() == 0)
              printf("Я дочерний процесс, PID= %d\n", getpid());
       else
              printf("Я родительский процесс, PID= %d\n", getpid());

       return 0;
}
```

При запуске fork-demo вы увидите оба сообщения: одно от родительского процесса, другое — от дочернего. В каком порядке будут выведены эти сообщения, сказать нельзя: процессы в Linux выполняются не одновременно, поэтому первым может появиться как сообщение родительского, так и дочернего процесса.

10.1.3. Семейство функций *exec*

Мы только что создали простейшее многопоточное приложение (конечно, это не совсем то, что вы ожидали, но потокам посвящена вся следующая глава, в которой мы все подробно и обсудим). Каждый поток может заниматься своими задачами. А теперь давайте рассмотрим классический запуск программы — когда одна программа запускает другую программу. Для этого нужно использовать одну из функций, описанных в unistd.h:

```
int execl (const char * PATH, const char * ARG, ...);
int execle (const char * PATH, const char * ARG, ..., const char ** ENVP);
int execlp (const char * FILE, const char * ARG, ...);
int execv (const char * PATH, const char ** ARGV);
int execvp (const char * FILE, const char ** ARGV);
int execve (const char * PATH, const char ** ARGV, const char ** ENVP);
```

Сначала рассмотрим классический запуск другой программы, а затем уж рассмотрим разницу между всеми этими системными вызовами (листинг 10.3).

Листинг 10.3. Запуск другой программы (exec.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int main (void)
{
        char * args[] = {
                "ls",
                "/",
                NULL
        };

        pid_t r = fork();

        if (r == 0) {
                execve("ls", args, NULL);
        }

        return 0;
}
```

Сначала мы создаем дочерний процесс, используя системный вызов `fork()`. Далее функция `execve()` вызывается уже в дочернем процессе и загружает в него программу `ls`. Массив `args` — это массив параметров, которые будут переданы запускаемой программе. Третий параметр — это переменные окружения. В данном случае мы не передаем окружение запускаемой программе (`NULL`).

Передать окружение дочернему процессу можно двумя способами: либо использованием внешней переменной `environ`, либо с помощью массива, содержащего переменные окружения. Рассмотрим первый способ (листинг 10.4).

Листинг 10.4. Передача окружения с помощью переменной `environ`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

extern char ** environ;

int main (void)
{
        char * args[] = {
                "ls",
                "/",
                NULL
        };

        pid_t r = fork();

        if (r == 0) {
                execve("ls", args, environ);
        }

        return 0;
}
```

Чтобы были заметны отличия от листинга 10.3, я их специально выделил полужирным. Сначала мы объявляем внешнюю переменную `environ`, а затем передаем ее функции `execve()`.

Теперь рассмотрим другой способ (листинг 10.5). Он заключается в том, что мы сами формируем массив, содержащий пары ПАРАМЕТР=ЗНАЧЕНИЕ.

Листинг 10.5. Передача окружения с помощью массива окружения

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
        char * args[] = {
                "program",
                "/",
                NULL
        };

        char * envp [] {
                "USER=den",
                "HOME=/home/den",
                NULL
        }

        execve("./program", args, envp);

        return 0;
}
```

Теперь приступим к изучению функций семейства `exec`. Все они используются для запуска новых процессов, разница только в наборе параметров. Начнем с функции `execve()`:

```
int execve (const char * PATH, const char ** ARGV, const char ** ENVP);
```

Данной функции нужно передать три параметра: путь к исполнимому файлу, массив аргументов и массив переменных окружения. Первый параметр не вызывает никаких вопросов — нужно указать путь запускаемой программе. Поиск программы возлагается полностью на программиста, поэтому лучше всего указать полный путь к программе, например /bin/bash.

Второй параметр — массив аргументов, которые будут переданы запускаемой программе. Первый параметр в этом массиве должен быть обязательно именем запускаемой программы, но не первым аргументом, который вы хотите передать программе. Последний параметр должен быть равен `NULL`. Например, вы хотите запус-

тить программу /usr/bin/program и передать ей параметры `arg1` и `arg2`. Массив параметров должен выглядеть так:

```
char * args[] = {
"/usr/bin/program",
"arg1",
"arg2",
NULL
};
```

Массив переменных окружения мы тоже уже рассматривали. Каждый элемент этого массива является строкой вида "ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ" *(см. главу 7)*. Последний элемент этого массива также должен быть равен `NULL`.

Функция `execl()` отличается от `execve()` тем, что ей передается не массив параметров программы, а просто список параметров. Каждый параметр этой функции (не считая первого) является параметром запускаемой программы. Передача переменных окружения не допускается (дочерний процесс получает окружение родительского процесса):

```
int execl (const char * PATH, const char * ARG, ...);
```

Пример:

```
execl("/usr/bin/program", "/usr/bin/program", "arg1", "arg2", NULL);
```

> ***Примечание***
>
> Необходимость передачи имени программы заключается не в особенностях функций, а в особенностях самой операционной системы. Нумерация параметров начинается с 0, параметр с номером 0 обычно содержит имя запускаемой программы. Теоретически вы можете в качестве параметра с номером 0 указать все, что вам угодно: если программа не обращается к этому параметру, ничего страшного не произойдет.

Функция `execlp()` похожа на `execl()`, разница заключается в том, что вы можете не указывать полный путь к программе, функция попытается найти программу в списке каталогов, хранящихся в переменной окружения `PATH`:

```
int execlp (const char * FILE, const char * ARG, ...);
```

Пример использования функции:

```
execlp("program", "program", "arg1", "arg2", NULL);
```

Функция `execle()` является аналогом функции `execl()`, но позволяет передать массив с переменными окружения:

```
int execle (const char * PATH, const char * ARG, ..., const char ** ENVP);
```

Функции `execvp()` и `execv()` похожи на `execve()`, обе передают параметры в программу, используя массив параметров, но обе не предусматривают передачи массива с переменными окружения. Разница между ними в том, что функция `execv()` требует передачи полного пути к исполняемому файлу программы, а функция `execvp()` позволяет указать только имя программы:

```
int execv (const char * PATH, const char ** ARGV);
int execvp (const char * FILE, const char ** ARGV);
```

Примеры использования функций:

```
execv("/usr/bin/program", args);
execv("program", args);
```

10.2. Системный вызов *wait()*: ожидание завершения дочернего процесса

Вспомним, как мы создали простейшее многопоточное приложение. Дочерний и родительский процессы выполняются непоследовательно и работают независимо друг от друга. Для многопоточного приложения это даже хорошо — дочерний процесс может заниматься своей задачей, не дожидаясь каких-либо инструкций от родительского процесса.

Но в некоторых случаях нужно организовать последовательное выполнение процессов: чтобы родитель дождался завершения своего потомка. Простейший пример такого поведения — командная оболочка. Командный интерпретатор ожидает завершения запущенной программы.

Самый простой способ организовать последовательное выполнение процессов — использовать функцию `system()`, которая была рассмотрена в предыдущей главе. Но есть более правильный способ — использование системного вызова `wait()`, который блокирует родительский процесс, пока не завершился дочерний.

Давайте рассмотрим пример использования `wait()`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <wait.h>
...
int pid;  unsigned short status;
             ...
    if((pid = fork()) == 0 ){
            /* Дочерний процесс */
            execl(....);
            perror("exec не удался"); exit(1);
    }
    /* Родительский процесс */
    while((pid = wait(&status)) > 0 )
      printf("Завершен дочерний процесс pid=%d с кодом %d\n",
               pid, status >> 8);
    printf( "Больше дочерних процессов нет\n");
```

Теперь разберемся, что к чему. Первым делом нам нужно подключить заголовочный файл wait.h. Затем функцией `fork()` мы создаем новый процесс. В дочернем процессе мы запускаем какую-нибудь программу функцией `execl()` или любой другой из семейства `exec`. Когда дочерний процесс будет завершен, он вернет какое-то значение — зависит от причины его завершения (обычно возвращается 0, если все прошло удачно). Если же нам не удалось запустить дочерний процесс, то мы используем функцию `perror()` для вывода сообщения об ошибке (хотя можно было бы использовать обычную `printf()` — как вам больше нравится). Поскольку процесс не был запущен, мы должны возвратить какое-то значение в родительский процесс, это мы делаем с помощью функции `exit()` — мы возвращаем значение 1.

В родительском процессе мы в цикле ожидаем завершения всех потомков (вдруг вы надумали запустить несколько нитей?). Системный вызов `wait()` возвращает PID дочернего процесса, а в переменную `status` записывается код завершения (либо номер сигнала, "убившего" процесс). После цикла мы выводим сообщение об отсутствии дочерних процессов.

Для обработки статуса дочернего процесса можно использовать несколько макросов. Рассмотрим тот же цикл `while()`, но в его теле мы будем обрабатывать код завершения дочернего процесса:

```
while((pid = wait(&status)) > 0){
        if( WIFEXITED(status)){
          printf( "Процесс %d завершен с кодом %d\n",
                 pid, WEXITSTATUS(status));
        } else
 if( WIFSIGNALED(status)){
          printf( "Процесс %d убит сигналом %d\n",
                  pid, WTERMSIG(status));
          if(WCOREDUMP(status))
          printf( "Core dumped\n" );
                  } else if( WIFSTOPPED(status)){
          printf("Процесс %d остановлен сигналом %d\n",
                 pid, WSTOPSIG(status));
        } else if( WIFCONTINUED(status)){
          printf( "Процесс %d продолжен\n",
                           pid);
        }
    }
```

В основном вы будете использовать следующие макросы:

- `WIFEXITED()` — возвращает ненулевое значение, если потомок завершился с помощью инструкции `return` в функции `main` или с помощью системного вызова `exit()`;
- `WEXITSTATUS` — возвращает код возврата завершенного процесса. Данный макрос нужно вызывать, только если `WIFEXITED()` вернул ненулевое значение;

- `WIFSIGNALED()` — возвращает ненулевое значение, если процесс был "убит" одним из сигналов;
- `WIFSTOPPED()` — возвращает ненулевое значение, если процесс был остановлен (но не завершен!) одним из сигналов;
- `WCOREDUMP()` — произошла критическая ошибка, процесс завершен некорректно, образовался дамп памяти `core`, который можно передать отладчику `gdb` для анализа ошибки;
- `WCONTINUED()` — выполнение процесса было продолжено сигналом.

Есть и другие макросы, но, как правило, они используется редко.

10.3. Обработка сигналов

Как было уж показано, каждый процесс может создать новый процесс, используя системный вызов `fork()`. С помощью системного вызова `wait()` родительский процесс может ожидать свои процессы-потомки. Запустить другую программу можно одной из функций `exec*()`.

Процесс-потомок может завершить свою работу с помощью системного вызова `exit()`. Каждый процесс реагирует на сигналы. Сигнал — это способ информирования процесса ядром о происшествии какого-то события. Родительский процесс может отправить сигнал дочернему процессу. Вообще говоря, процесс может отправить любой сигнал любому процессу, для этого нужно только знать PID этого процесса и обладать необходимыми полномочиями. Если процесс A запущен от имени den, то он может послать сигнал любому процессу, запущенному от имени этого пользователя, но не процессу, который запущен от имени другого пользователя. Если же процесс запущен от имени root, то он может послать сигнал любому процессу в системе.

Процесс может установить реакцию на любой сигнал, для этого используется системный вызов `signal()`:

```
signal(snum, function);
```

Первый параметр — номер сигнала или его название (см. далее), второй параметр — функция, которая будет запущена для обработки сигнала. Вместо функции можно указать `SIG_IGN`, если нужно проигнорировать сигнал. Однако не все сигналы можно игнорировать.

Примеры вызова `signal()`:

```
#include <signal.h>
...
signal(SIGTTIN, SIG_IGN);
signal(SIGHUP, signal_handler);
```

Послать сигнал процессу можно с помощью системного вызова `kill()`:

```
kill(pid, snum);
```

Первый параметр — это PID процесса, которому отправляется сигнал, второй — номер (или имя сигнала). Полный список сигналов вы найдете в заголовочном файле signal.h. Рассмотрим самые интересные сигналы.

- SIGHUP (номер 1) — предназначен для того, чтобы информировать программу о потере связи с управляющим терминалом. Ранее терминалы подключались к системе с помощью модемов, поэтому название сигнала происходит от англ. *hung up* — повесить трубку. По умолчанию программа, получившая этот сигнал, завершает работу. Но вы можете установить обработчик этого сигнала. Программы-демоны, получив этот сигнал, обычно перезапускаются.
- SIGINT (номер 2) — посылается процессу, если пользователь терминала (обратите внимание: этот сигнал относится к программам, запущенным в терминале) нажал сочетание клавиш <Ctrl>+<C> (запрос на прерывание процесса).
- SIGKILL (номер 9) — завершает работу программы, игнорировать или обрабатывать этот сигнал нельзя.
- SIGSEGV (номер 11) — сигнал отправляется процессу, который попытался обратиться к области памяти, которая ему не принадлежит. Если обработчик сигнала не установлен, программа завершает работу, на диске сохраняется образ памяти (core).
- SIGTERM (номер 15) — "вежливое" завершение программы. Получив этот сигнал, программа может освободить занятые ресурсы (закрыть соединения, файлы). Получение этого сигнала означает, что пользователь (или операционная система) хочет завершить работу программы.
- SIGCHLD (номер 17) — посылается процессу, если дочерний процесс завершился. По умолчанию этот сигнал игнорируется.
- SIGCONT (номер 18) — возобновляет выполнение процесса, который был остановлен сигналом SIGSTOP.
- SIGSTOP (номер 19) — приостанавливает выполнение процесса. Данный сигнал нельзя перехватить или проигнорировать.
- SIGTSTP (номер 20) — приостанавливает процесс по команде пользователя, обычно для этого используется комбинация клавиш <Ctrl>+<Z>.

10.4. Получение информации о процессе

С процессом тесно связаны три идентификатора:

- PID (Process ID) — идентификатор процесса. Уникален для каждого процесса;
- PPID (Parent PID) — идентификатор родительского процесса;
- UID (User ID) — идентификатор (номер) пользователя, запустившего процесс.

Получить все эти идентификаторы можно с помощью системных вызовов, которые объявлены в заголовочном файле unistd.h:

```
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid(void);
```

Типы данных `pid_t` и `uid_t` объявлены в заголовочном файле sys/types.h. Другими словами, чтобы получить PID процесса, нужно подключить два заголовочных файла:

```
#include <unistd.h>
#include <sys/types.h>
```

Заметьте, что функция `getuid()` возвращает числовой идентификатор пользователя. Чтобы узнать имя пользователя (оно хранится в /etc/passwd), нужно использовать функцию `getpwuid()`, объявленную в файле pwd.h:

```
struct passwd * getpwuid (uid_t UID);
```

Сама структура `passwd` нас не интересует, нас интересует только поле `pw_name`, содержащее имя пользователя:

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
...
// Получаем UID пользователя
uid_t uid = getuid();
// Получаем имя пользователя
struct passwd * p = getpwuid(uid);
if (p == NULL)
{
printf("Error\n");
return 1;
}
printf("Username: %s", p->pw_name);
```

ГЛАВА 11

Многопоточные приложения

11.1. Введение в потоки

Как мы уже знаем, потоки (нити, threads) используются для параллельного выполнения некоторых задач в рамках одной программы. Типичный пример многопоточного приложения, с которым работал любой пользователь, — менеджер закачки файлов. Загружаемый файл "виртуально" разбивается на несколько частей, затем запускается несколько потоков и каждый поток загружает свою часть файла, что позволяет сократить время загрузки всего файла.

В предыдущей главе мы создали что-то наподобие многопоточного приложения, однако для достижения нашей цели мы использовали системный вызов `fork()`, который порождает параллельный процесс, но не поток в классическом понимании этого слова.

Системный вызов `fork()` порождал полностью независимый процесс, который мог существовать, даже если родительский процесс завершил свою работу. У дочернего процесса был свой PID и полностью свои данные. Чтобы продемонстрировать это, модифицируем программу, представленную в листинге 10.2. Новая версия программы представлена в листинге 11.1.

Листинг 11.1. Пример: процессы vs потоки

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main (void)
{
        int count = 0;

        if (fork() == 0) {
        count++;
```

```
                printf("Дочерний процесс, count = %d\n", count);
                return 0;
                }
        else
                printf("Родительский процесс, count = %d\n", count);

        return 0;
}
```

После выполнения этой программы вы поймете, что переменная `count` родительского процесса не имеет никакого отношения к переменной `count` дочернего процесса: ведь каждый процесс в Linux имеет собственное адресное пространство.

Да, существуют способы обмена данными между процессами (IPC), которые будут рассмотрены в следующей главе. Однако средства межпроцессного взаимодействия требуют особого внимания программиста: как минимум, злоумышленник может найти уязвимость в вашей программе, если вы плохо организуете межпроцессное взаимодействие. Да и сам процесс организации такого взаимодействия не очень простой, гораздо проще использовать механизм потоков.

Как уже было сказано ранее, у каждого процесса есть свой идентификатор процесса (PID). У каждого потока есть идентификатор потока (THREAD_ID), но каждый поток выполняется в рамках одного процесса. Если один из потоков завершает программу, то будут прекращены все потоки сразу. Это еще одно отличие от процессов: завершение дочернего процесса никак не повлияет на родительский и наоборот.

В Linux потоки выполняются независимо — как и процесс, однако не забывайте о том, что потоки выполняются в рамках одного процесса. Организовать поток в Linux очень и очень просто:

- нужно создать функцию, которая потом станет функцией потока;
- функция `pthread_create()` создает поток, для каждого потока назначается своя потоковая функция. После создания потоков их функции будут выполняться параллельно.

Ваша программа продолжает выполняться сразу после вызова функции `pthread_create()`. Основная программа не ждет завершения потоковой функции.

Потоки в Linux реализованы в библиотеке `pthread`. Чтобы подключить ее к программе, нужно скомпилировать последнюю с аргументом `-lpthread`.

11.2. Функция *pthread_create()*

Рассмотрим функцию `pthread_create()`:

```
int pthread_create(pthread_t * THREAD_ID, void * ATTR,
        void *(*THREAD_FUNC) (void*), void * ARG);
```

Первый параметр задает переменную, в которую будет записан идентификатор нового потока. Второй параметр — это атрибуты потока. Потоки в Linux — довольно сложная тема, достойная отдельной книги, поэтому в данной книге мы для упрощения восприятия вообще не будем передавать какие-либо аргументы нашим потокам — просто указывайте `NULL` в качестве второго параметра.

Третий параметр — это потоковая функция, а четвертый — аргументы, которые будут переданы этой функции.

Для закрепления прочитанного материала давайте напишем программу, порождающую два потока (листинг 11.2).

Листинг 11.2. Программа threads.c

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>          /* Подключаем для работы с потоками */

void * thread1 (void)
{
       printf("\nЯ — поток 1\n");
       sleep(5);              /* Засыпаем на 5 секунд */
}

void * thread2 (void)
{
       printf("\nЯ — поток 2\n");
       sleep(10);             /* Засыпаем на 10 секунд */
}

int main (void)
{
       pthread_t tid1;        /* Идентификатор первого потока */
       pthread_t tid2;        /* Идентификатор второго потока */

       /* Запускаем первый поток */
       pthread_create (&tid1, NULL, &thread1, NULL);
       /* Запускаем второй поток */
       pthread_create (&tid2, NULL, &thread2, NULL);

       while (1);     /* бесконечный цикл */
       return 0;
}
```

Наша простая программа запускает два потока с потоковыми функциями `thread1()` и `thread2()`. Потоковые функции не подразумевают передачу параметров, поэтому в качестве четвертого аргумента функции `pthread_create()` указан `NULL`. После вы-

зова функций `pthread()` все три функции — `main()`, `thread1()` и `thread2()`, будут выполняться одновременно.

Особого внимания заслуживает бесконечный цикл `while()`. Он нужен для того, чтобы основная функция `main()` не завершилась раньше потоковых функций. Прервать выполнение программы можно с помощью комбинации клавиш <Ctrl>+<C>. Сейчас такой выход из программы выглядит довольно варварским, но чуть позже мы разберемся, как избавиться от бесконечного цикла.

Напомню, что компилировать программу нужно с опцией `-lpthread`. При компиляции программы возможны предупреждения о том, что тип аргумента, передаваемый функции `pthread_create()`, не соответствует ожидаемому (однако программа откомпилируется и будет работать). Правильнее потоковые функции объявлять так:

```
void * thread1 (void * args)
...
void * thread2 (void * args)
```

Однако поскольку я не планировал передавать функциям аргументы, то и объявил их так, как показано в листинге 11.2. Проведем еще один эксперимент с потоковыми функциями. Измените код файла threads.c так:

```
void * thread1 (void)
{
       printf("\nЯ — поток 1\n");
       sleep(5);              /* Засыпаем на 5 секунд */
       return NULL;
}

void * thread2 (void)
{
       sleep(10);             /* Засыпаем на 10 секунд */
       printf("\nЯ — поток 2\n");
       return NULL;
}
```

В этом случае тоже будут запущены два потока. Первый поток сразу выведет сообщение и "заснет" на 5 секунд. Второй поток сначала будет ждать 10 секунд, а потом выведет сообщение. Несмотря на то что завершился первый поток, второй продолжает выполняться, как и основная программа. Кстати, еще одно "косметическое" изменение: поскольку у наших функций тип `void`, то для большей корректности нужно, чтобы они возвращали значение `NULL`.

А теперь попробуем изменить эти функции так:

```
void * thread1 (void)
{
       printf("\nЯ — поток 1\n");
       sleep(5);              /* Засыпаем на 5 секунд */
       exit(1);
```

```
}

void * thread2 (void)
{
       sleep(10);            /* Засыпаем на 10 секунд */
       printf("\nЯ — поток 2\n");
       return NULL;
}
```

Компилятор "выругается", что использование `exit()` не предусмотрено в такого рода функциях, но тем не менее скомпилирует программу. Запустите ее. Вы увидите только сообщение первого потока, потом программа подождет 5 секунд и завершит свою работу. Сообщение второго потока вы так и не увидите. Как видите, если один из потоков завершил работу программы, то завершаются сразу все другие потоки, запущенные этим процессом.

11.3. Передача аргументов потоковой функции

В программе threads.c мы не передавали аргументы нашим потоковым функциям, однако в реальных проектах без аргументов не обойтись. Модифицируем программу так, чтобы она передавала параметры в потоковые функции (листинг 11.3).

Листинг 11.3. Передача аргументов в потоковые функции

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>         /* Подключаем для работы с потоками */

void * thread1 (void * arg)
{
       int c = *(int*) arg;
       printf("\nThread 1: %d\n", c);
       sleep(5);             /* Засыпаем на 5 секунд */
       return NULL;
}

void * thread2 (void * arg)
{
       int c = *(int*) arg;
       printf("\nThread 2: %d\n", c);
       sleep(10);            /* Засыпаем на 10 секунд */
       return NULL;
}

int main (void)
```

```
{
        pthread_t tid1;        /* Идентификатор первого потока */
        pthread_t tid2;        /* Идентификатор второго потока */
        int t_arg;

        t_arg = 10;            /* Будет передано в первый поток */
        /* Запускаем первый поток */
        pthread_create (&tid1, NULL, &thread1, &t_arg);

        t_arg = 50;            /* Будет передано во второй поток */
        /* Запускаем второй поток */
        pthread_create (&tid2, NULL, &thread2, &t_arg);

        while (1);     /* Бесконечный цикл */
        return 0;
}
```

Результат выполнения нашей модифицированной программ изображен на рис. 11.1. Запускаются оба потока, и каждый выводит переданное ему значение.

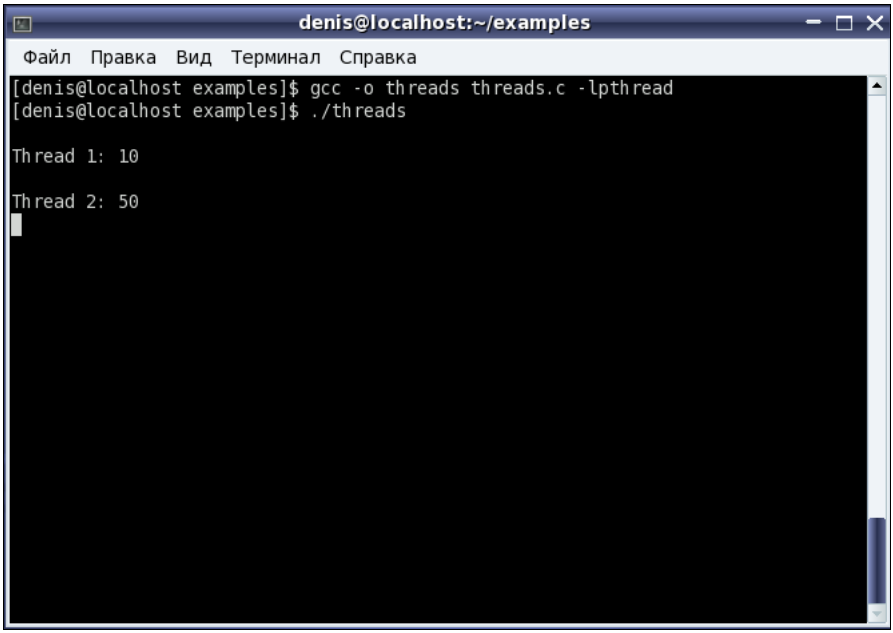


Рис. 11.1. Потокам переданы аргументы

Приведенный способ позволяет передать в функцию только один аргумент. Если же нужно передать несколько аргументов, используйте структуры. Напишем небольшую программу, создающую один поток (да, всего один — для упрощения ее кода) и передающую ему несколько аргументов (листинг 11.4).

Листинг 11.4. Передача потоковой функции нескольких аргументов

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>          /* Подключаем для работы с потоками */

struct t_args
{
       int arg1;
       int arg2;
}

void * thread1 (void * arg)
{
       struct t_args t = *(struct t_args*) arg;
       printf("\nThread 1: %d %d\n", t.arg1, t.arg2);
       sleep(5);              /* Засыпаем на 5 секунд */
       return NULL;
}

int main (void)
{
       pthread_t tid1;        /* Идентификатор потока */
       struct t_args ta;

       /* Заполняем структуру аргументов потоковой функции */
       ta.arg1 = 10;
       ta.arg2 = 50;


       /* Запускаем первый поток */
       pthread_create (&tid1, NULL, &thread1, &ta);


       while (1);     /* Бесконечный цикл */
       return 0;
}
```

Как видите, ничего сложного в этом нет.

11.4. Правильное завершение потока: функция *pthread_exit()*

Обычная программа может завершаться либо возвратом (`return`) из функции `main`, либо через вызов `exit()`. К чему приводит вызов `exit()` из потоковой функции, мы уже знаем. Потоки можно завершать возвратом из потоковой функции, но коррект-

нее для этого использовать функцию pthread_exit(), объявленную в заголовочном файле pthread.h:

```
void pthread_exit (void * RESULT);
```

Аргумент RESULT — это возвращаемое потоком значение. Дальше мы рассмотрим, как можно передавать данные из потока в основной процесс или в другой поток. Вот как выглядит потоковая функция с использованием функции pthread_exit():

```
void * thread1 (void * arg)
{
        struct t_args t = *(struct t_args*) arg;
        printf("\nThread 1: %d %d\n", t.arg1, t.arg2);
        sleep(5);              /* Засыпаем на 5 секунд */
        pthread_exit(NULL);
}
```

11.5. Избавляемся от бесконечного цикла: функция *pthread_join()*

До этого момента у нашего многопоточного приложения было два существенных недостатка. Первый — это варварский цикл while, второй — то, что наши потоки не возвращали никаких значений в основной процесс. Ведь потоки используются обычно для параллельных вычислений. Запускать потоки и передавать в них параметры мы уже научились, но пока не умеем получать результат выполнения потока.

С циклом while() разобраться достаточно просто: можно объявить глобальную переменную и в цикле while контролировать ее значение. Поток перед завершением бы устанавливал эту переменную, скажем, в 1. В цикле while проверялось бы: если переменная равна 1, то производится выход из цикла (break). Аналогично, можно и передать результат выполнения. Рассмотрим листинг 11.5.

Листинг 11.5. "Правильный" цикл while

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int end = 0;          /* Некоторая глобальная переменная */

/* Обычная потоковая функция, которой передается аргумент */
void * thread1(void * arg)
{
        int c = *(int*) arg;
        printf("\nThread 1: %d\n",c);
        sleep(5);
```

```
        /* Функция устанавливает глобальную переменную в 1 */
        end = 1;
        return NULL;
}

int main (void)
{

        pthread_t tid1;
        int t_arg;

        t_arg = 10;
        /* Создаем поток */
        pthread_create(&tid1, NULL, &thread1, &t_arg);

        /* Цикл, пока end не равна 1. Как только поток
        завершит вычисления, он установит переменную end
        в 1 и программа завершит работу */
        while (end!=1);

        return 0;
}
```

Логика программы проста: поток устанавливает глобальную переменную, которую мы используем как признак окончания работы потока — как только поток ее установит, мы завершим работу программы.

Все бы хорошо и мы бы так и продолжали писать наши приложения с использованием цикла `while`, если бы у нас не было функции `pthread_join()`. Мы только что изобрели велосипед, но могли бы этого и не делать, зная о существовании функции `pthread_join()`.

Функция `pthread_join()` позволяет "подключиться" к потоку. Данную функцию можно вызвать, например, из основного процесса для подключения к потоку и получения возвращаемого потоком значения:

```
int pthread_join (pthread_t THREAD_ID, void ** DATA);
```

Функция блокирует программу, пока не завершится поток с идентификатором `THREAD_ID`. Второй параметр будет содержать результат выполнения потока, установленный функцией `pthread_exit()`.

Вы можете вызвать функцию не только из основной программы, но и из другого потока — функция заблокирует вызывающий поток, пока не будет завершен вызываемый поток. Учитывая эту особенность функции `pthread_join()`, вы можете синхронизировать потоки.

Разберемся, как использовать `pthread_join()`. Сначала вы запускаете несколько потоков, как было показано ранее. Затем основная программа может делать все что угодно: она может выполнять какие-нибудь полезные действия, ведь ее выполнение

после запуска потоков не приостанавливается. Когда ей нужно будет получить значение, возвращенное потоком, вы вызываете `pthread_join()`. Не нужно вызывать `pthread_join()` сразу после создания потока, иначе программа будет сразу же заблокирована — она будет ждать завершения потока.

Если на момент вызова `pthread_join()` поток еще не завершил работу, ваша программа или другой поток будет ожидать завершения потока. А если поток уже был завершен на момент вызова `pthread_join()`, тогда вызывающая сторона попросту получит возвращенное потоком значение.

Напишем простейшую программу, которая создает поток и передает ему аргумент. Поток вычисляет квадрат аргумента и возвращает его функцией `pthread_exit()`. Основной процесс "подключается" к потоку и получает результат вычислений, затем выводит его на стандартный вывод (листинг 11.6).

Листинг 11.6. Передача результата выполнения потока

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void * thread1(void * arg)
{
       int c = *(int*) arg;          /* Получаем аргумент */
       c = c * c;                    /* Производим вычисления */
       sleep(5);                     /* Для создания эффекта "длительных"
                                     вычислений */

       /* Возвращаем результат вычислений */
       pthread_exit((void *)c);
}

int main (void)
{

       pthread_t tid1;       /* Идентификатор потока */

       /* t_arg — аргумент, передаваемый потоковой функции,
       t_res — в эту переменную будет сохранен результат потоковой
       функции */
       int t_arg, t_res;

       t_arg = 10;

       /* Запускаем поток */
       pthread_create(&tid1, NULL, &thread1, &t_arg);
```

```
        /* Получаем результат потоковой функции */
        pthread_join(tid1,(void *) &t_res);

        /* Выводим результат */
        printf("\nResult = %d\n", t_res);

        return 0;
}
```

11.6. Получение информации о потоке

Наверняка вы помните функцию `getpid()`, возвращающую идентификатор текущего процесса. Внутри потоковой функции вы можете использовать функцию `pthread_self()`, которая возвращает идентификатор текущего потока:

```
pthread_t pthread_self (void);
```

Зачем потоку знать свой идентификатор? Все очень просто: чтобы избежать зацикливания программы, когда поток вызывает функцию `pthread_join()` для самого себя. Для сравнения идентификаторов потоков лучше использовать не оператор ==, а функцию `pthread_equal()`:

```
if (!pthread_equal(pthread_self(), tid))
        pthread_join(tid, &t_res);
```

11.7. Прерывание потока

Равно как любой поток может "подключиться" к любому другому потоку одного процесса, так и любой поток может отменить другой поток с помощью функции `pthread_cancel()`:

```
int pthread_cancel(pthread_t THREAD_ID);
```

В случае ошибки функция возвращает ненулевое значение или 0, если поток завершен.

Однако вызов `pthread_cancel()` не означает немедленного удаления потока — это только лишь отправка запроса на удаление.

Как уже было отмечено, потоки в Linux заслуживают отдельной книги. В рамках этой книги мы рассмотрели потоки достаточно подробно. Дополнительную информацию вы можете получить в справочной системе:

```
man pthreads
man pthread_create
man pthread_cancel
man pthread_join
man pthread_exit
man pthread_equal
man pthread_self
```

ГЛАВА 12

Взаимодействие процессов

12.1. Способы взаимодействия

Процессы могут "общаться" между собой, т. е. обмениваться информацией. Процесс-родитель может передать дочернему процессу какую-либо информацию, а тот уже будет ее обрабатывать. Взаимодействие процессов называется IPC — Inter-Process Communication.

В Linux возможны следующие способы взаимодействия процессов:

- каналы;
- именованные каналы типа FIFO (First In First Out);
- очереди сообщений;
- семафоры;
- разделяемые сегменты памяти;
- сетевые сокеты.

Если вы знакомы с историей UNIX, то знаете, что со времен еще первых версий UNIX было два способа взаимодействия: System V и BSD. Семафоры, разделяемая память и очередь сообщений — эти средства появились в операционной системе System V компании AT&T. В Linux поддерживаются оба типа взаимодействия процессов.

12.2. Каналы

Знакомство со способами взаимодействия процессов мы начнем с самого старого способа взаимодействия процессов — каналов. Каналы появились еще в самых первых версиях операционной системы UNIX.

Каналы бывают *полудуплексными* и *полнодуплексными* (каналы потоков). Полудуплексные каналы позволяют обмениваться информацией только в одном направлении, например когда родительский процесс передает информацию на стандартный

ввод дочернего процесса. Полнодуплексные каналы позволяют обмениваться информацией в обоих направлениях. Полнодуплексные каналы не будут рассмотрены в этой книге.

В *главе 8* было рассмотрено перенаправление ввода/вывода. Сейчас нас интересует возможность перенаправления стандартного вывода одной программы на стандартный вывод другой, например:

```
cat some_big_file | less
```

Перенаправление ввода/вывода можно осуществить и программным путем, т. е. без вмешательства пользователя. Программа может перенаправить свой вывод на стандартный ввод другой программы. Реализовать ввод/вывод между процессами можно с помощью функции `popen()`:

```
FILE * popen(const char *command, const char *type);
```

Первый параметр — это название программы, которую мы хотим запустить (это и будет наш дочерний процесс). Второй параметр определяет тип доступа. Установите значение `r`, если вам нужно читать вывод дочернего процесса; если же вам нужно передать информацию на стандартный ввод порожденного процесса, установите значение `w`.

Вы не можете установить комбинированный режим `wr`, который, по логике вещей, должен разрешить двусторонний обмен, поскольку полудуплексные каналы подразумевают обмен информации в одном направлении.

Функция `popen()` возвращает указатель `FILE` или пустой указатель `NULL`, если вызов не удался. Канал закрывается вызовом функции `pclose()` после завершения операций ввода/вывода. Во время работы с каналом рекомендуется использовать вызов `fflush()`, чтобы предотвратить задержки из-за буферизации.

Рассмотрим самый простой пример использования функции `popen()` — листинг 12.1. Используя механизм каналов, мы будем читать вывод команды `ls` и отображать его на экране. Вместо `ls` вы можете указать любую другую команду. Вывод команды `ls` читается построчно, мы можем обработать каждую строку вывода команды.

Листинг 12.1. Пример использования `popen()`

```
#include <stdio.h>

int main() {
  FILE *in;
  extern FILE *popen();
  char buff[512];

  /* Создаем канал, вызываем программу ls, режим "r" */
  if (!(in = popen("ls", "r"))) {
    exit(1);
  }
```

```
  /* Читаем вывод ls и отображаем его на экране */
  while (fgets(buff, sizeof(buff), in) != NULL ) {
    printf("Вывод: %s\n", buff);
  }

  /* Закрываем канал */
  pclose(in);
}
```

Приведенный пример довольно тривиален, но он демонстрирует, как с помощью каналов очень просто читать вывод дочернего процесса. Теперь рассмотрим более серьезный проект. У нас будет две программы — parent (родительский процесс) и child (дочерний процесс). Родительский процесс будет передавать информацию дочернему процессу, который, в свою очередь, прочитает информацию, обработает ее и выведет на стандартный вывод.

Исходный код родительского процесса представлен в листинге 12.2.

Листинг 12.2. Программа parent.c

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
       char buff[1024]={0};
       FILE * child;
       int status;

        /* Открываем канал для записи. Дочерний процесс —
              /usr/local/bin/child   */
        child = popen("/usr/local/bin/child", "w");
          if (!child)
           {
             printf("Ошибка при открытии канала\n");
             exit(1);
           }

        printf("Введите строку для передачи дочернему процессу: " );

        /* Читаем ввод пользователя */
        fgets(buff, sizeof (buff), stdin);

        /* Передаем данные дочернему процессу */
        fprintf(child, "%s\n", buff);
```

```
        /* Пересылаем содержимое буфера в канал */
        fflush(child);

        // Закрываем канал и проверяем состояние вызова pclose()
        status=pclose(child);

        if (!WIFEXITED(status))
                printf("Не могу закрыть канал\n");

        printf("Родительский процесс завершен\n");

       return 0;
}
```

На этот раз мы открываем канал для записи (тип доступа — `w`). Затем мы читаем со стандартного ввода информацию от пользователя и передаем ее дочернему процессу, который нам еще предстоит создать (листинг 12.3). После того как вы создадите и скомпилируете child.c, не забудьте поместить исполнимый файл child в /usr/local/bin. Программа child.c предельно проста: она читает со стандартного ввода строку и выводит ее на экран. Обратите внимание на листинг 12.2. В нем есть один момент — вызов функции `fflush()`. Она "выталкивает" содержимое из буфера в дочерний процесс. Использование `fflush()` позволяет избежать задержек при передаче информации в дочерний процесс.

Листинг 12.3. Программа child.c

```
#include <stdio.h>

int main()
{
       char buff[1024]={0};
       fgets(buff, sizeof (buff), stdin);
       printf("Прочитана строка: %s\n",buff);
       printf("Дочерний процесс завершил работу\n");
       return 0;
}
```

Скомпилируйте обе программы:

```
gcc -o parent parent.c
gcc -o child child.c
```

Скопируйте child в /usr/local/bin:

```
sudo cp ./child /usr/local/bin/child
```

Запустите ./parent. Вывод на экране будет примерно таким (123456 — введенное пользователем значение для передачи дочернему процессу):

```
Введите строку для передачи дочернему процессу: 123456
Прочитана строка: 123456
Дочерний процесс завершил работу
Родительский процесс завершен
```

12.3. Именованные каналы типа FIFO

Следующий способ взаимодействия процессов — каналы FIFO (First In First Out). Такие каналы организованы по принципу очереди: "первый вошел, первый вышел". Канал FIFO существенно отличается от обычного канала, который был рассмотрен в предыдущем разделе:

- канал FIFO сохраняется в файловой системе в виде файла, поэтому такие каналы называются именованными;
- с именованным каналом могут работать все процессы, а не только родительский и дочерний: ведь к FIFO-каналу можно обратиться как к обычному файлу;
- именованный канал остается в файловой системе даже после завершения обмена данными. При следующем использовании канала его не нужно заново создавать.

Итак, мы выяснили самое важное отличие именованного канала от обычного полудуплексного: FIFO-канал находится в файловой системе, а полудуплексный — просто в оперативной памяти.

Создать FIFO-канал можно с помощью команд оболочки и с помощью системного вызова `mknod()`. Рассмотрим первый вариант:

```
sudo mknod FIFO p
sudo mkfifo a=rw FIFO
```

Обе команды создают канал с именем FIFO. Вместо команды `mknod` можно использовать системный вызов с таким же названием:

```
int mknod( char *pathname, mode_t mode, dev_t dev );
```

Функция `mknod()` может создать любой файл, а не только канал, например устройство, файл. Ранее мы с помощью команды `mknod` создали FIFO-канал, сейчас мы создадим такой же канал с помощью вызова `mknod()`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
...

mknod("FIFO", S_FIFO|0666, 0);
```

Первый параметр задает имя FIFO-канала или устройства — в зависимости от того, что вы хотите создать. Второй параметр как раз и определяет тип создаваемого объекта. Вообще, второй параметр представляет собой комбинацию типа объекта и прав доступа к нему.

Тип объекта может быть следующим:

- `S_IFREG` — регулярный (обычный файл);
- `S_IFCHR` — символьное устройство;
- `S_IFBLK` — блочное устройство;
- `S_IFIFO` — именованный канал;
- `S_IFSOCK` — сокет.

Функция возвращает 0, если создание узла прошло успешно, или –1, если произошла ошибка. Проанализировать ошибку можно с помощью переменной `errno`, которая равна:

- `EFAULT`, `ENOTDIR`, `ENOENT` — неправильный путь;
- `EACCESS` — у вас недостаточно прав;
- `ENAMETOOLONG` — слишком длинный путь.

Об остальных значениях переменной `mknod` вы сможете узнать в `man 2 mknod`. Третий параметр (`dev`) задает имя вспомогательного устройства, но при создании каналов этот параметр неактуален.

Ранее мы создали канал с именем FIFO в текущем каталоге и установили права доступа 0666.

> ***Права доступа и umask***
>
> При программировании в Linux следует помнить одну очень важную особенность файловой системы, а именно маску прав доступа. Как мы уже знаем, для задания маски прав доступа используется системный вызов `umask()`. Когда вы в своей программе задаете права доступа к файлу (обычному файлу, устройству, каталогу, каналу — не важно), вы должны сначала вызвать `umask(0)`, а затем установить права доступа. Только в этом случае вы установите истинное значение прав доступа. Если маска прав доступа не равна 0, то устанавливаемые права доступа могут отличаться от тех, которые будут реально установлены:
>
> ```
> umask(0);
> mknod("FIFO", S_FIFO|0666, 0);
> ```

Настало время практики. Мы напишем две программы, демонстрирующие работу с именованными каналами. Первая программа создаст FIFO-канал и будет ожидать записи в него, а вторая программа как раз и произведет запись (но будет ждать, пока первый процесс не прочитает эти данные). Вторая программа завершит свою работу сразу после записи в канал, а вот первая будет содержать бесконечный цикл, поэтому завершить ее работу можно будет с помощью <Ctrl>+<C>.

Исходный код первой программы представлен в листинге 12.4, второй — в листинге 12.5.

Листинг 12.4. Программа fifo-read.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
```

```
#include <unistd.h>
#include <linux/stat.h>

/* Имя канала */
#define FIFO   "FIFO"

void main(void)
{
   FILE *fp;
   /* Буфер для чтения данных из канала */
   char buf[128];

   /*  Создаем канал FIFO с правами доступа 0666.
   Канал будет создан, только если он еще не существует*/
   umask(0);
   mknod(FIFO, S_IFIFO|0666, 0);

/* Ожидаем данные */
   while(1)
   {
      fp = fopen(FIFO, "r");
      fgets(buf, 128, fp);
      printf("Прочитано: %s\n", buf);
      fclose(fp);
   }

}
```

Теперь рассмотрим листинг 12.5. Напомню, что эта программа не завершит свою работу, пока первая программа не прочитает данные. Поэтому сначала нужно запустить первую программу, а затем программу из листинга 12.5.

Листинг 12.5. Программа fifo-write.c

```
#include <stdio.h>
#include <stdlib.h>

#define FIFO   "FIFO"

void main(int argc, char *argv[])
{
   FILE *fp;

   if ( argc != 2 ) {
      printf("Использование: fifo-write <строка>\n");
      exit(1);
      }
```

```
    fp = fopen(FIFO, "w");
    fputs(argv[1], fp);
    fclose(fp);
}
```

Скомпилируйте программы:

```
gcc -o fifo-read fifo-read.c
gcc -o fifo-write fifo-write.c
```

Запустите программу чтения канала:

```
./fifo-read
```

Перейдите на другой терминал (<Alt>+<F2>, если вы работаете в консоли) и запустите программу fifo-write:

```
./fifo-write test
```

Перейдите на первый терминал (где запущен процесс fifo-read), вывод на экране будет таким:

```
Прочитано: test
```

А что делать, если у нас есть несколько процессов, желающих записать в канал? Все нормально: fifo-write получает эксклюзивный доступ к каналу и блокирует канал до тех пор, пока не запишет данные. Остальные процессы, желающие записать данные в канал, ожидают окончания записи, а затем записывают данные в порядке очереди.

Может и возникнуть другая нештатная ситуация: если программа записала в канал, у которого нет "читателя", т. е. программа fifo-read не запущена, тогда "писателю" (программе fifo-write) будет послан сигнал `SIGPIPE`.

12.4. Очереди сообщений

12.4.1. Межпроцессное взаимодействие System V

Прежде чем приступить к рассмотрению очередей сообщений, нужно изучить теорию, а именно основы межпроцессного взаимодействия (IPC) System V. Как уже было отмечено, в этой модели IPC появились очереди сообщений, семафоры и разделяемые сегменты памяти. Каждый из этих объектов имеет уникальный идентификатор, позволяющий ядру однозначно идентифицировать объект. Идентификатор уникален только для объектов данного типа, а не для всей системы, т. е. в вашей системе может быть очередь сообщений с номером 123 и семафор с таким же номером, но не может быть двух очередей или двух семафоров с номером 123.

Чтобы получить уникальный идентификатор, системе нужен ключ (IPC Key). Данный ключ согласовывается с процессом-сервером и процессом-клиентом. Ключ генерируется не системой, а вашим приложением с помощью функции `ftok()`:

```
# include <sys/types.h>
# include <sys/ipc.h>
key_t ftok( char *pathname, char proj );
```

В случае ошибки эта функция возвращает –1, а в случае успеха — сгенерированный ключ. Сгенерированные ключи могут повторяться, поэтому после получения нового ключа проверьте, не используется ли он.

Ключ генерируется по номеру `inode` первого аргумента (нужно задать только имя файла, а `inode` будет вычислен функцией автоматически) и символа, заданного вторым аргументом. Это хоть как-то может гарантировать уникальность ключа, но теоретически ключи могут повторяться. Пример создания ключа:

```
key_t key_ipc;
key_ipc = ftok ("/etc/passwd", 'd');
```

Вы можете просмотреть статус всех объектов IPC с помощью команды `ipcs`. В данный момент, скорее всего, в вашей системе нет ни одного из объектов IPC, поэтому вывод будет таким:

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

Для удаления объекта IPC используется команда

```
ipcrm <msg | sem | shm> IPC_ID
```

Первый аргумент задает тип объекта (`msg` — очередь сообщений, `sem` — семафор, `shm` — сегмент памяти), второй — его идентификатор.

12.4.2. Структуры ядра для работы с очередями

Очередь сообщений — это связный список сообщений, находящийся в адресном пространстве ядра. Поскольку очередь сообщений является объектом IPC, то у каждой очереди есть свой уникальный идентификатор.

Структура ядра `msgbuf` (объявлена в файле /usr/src/linux/include/linux/msg.h) является буфером сообщений:

```
struct msgbuf {
    long mtype;          /* Тип сообщения */
    char mtext[1];       /* Текст сообщения */
};
```

Структура проста: ее поля могут хранить тип сообщения и текст сообщения. Тип сообщения позволяют хранить в очереди сообщения нескольких типов, что позво-

ляет использовать одну и ту же очередь, а не создавать несколько — по одной для сообщений каждого типа — это удобно. А текст сообщения, состоящий всего из одного символа, подойдет только в самых простых случаях. В своей программе лучше переопределить структуру `msgbuf` так:

```
struct my_buf {
long mtype;
       char mtext[256];
}
```

При желании в своей программе вы можете даже добавить дополнительные поля в эту структуру, например:

```
struct my_buf {
long mtype;
       char mtext[256];
       long status;
}
```

Максимальный размер сообщения задан в файле /usr/src/linux/include/linux/msg.h:

```
#define MSGMAX 4056
```

Также можно узнать максимальный размер сообщения с помощью `sysctl` (параметр `kernel.msgmax`). Помните, что параметр `msgmax` задает максимальный размер всего сообщения, а не только текстового поля. В нашем случае у нас есть два поля типа `long` (4 + 4 = 8 байтов) и поле `mtext` длиной 256 байтов, общий размер поля — 264 байта.

Ваши сообщения хранятся ядром в структуре `msg` (см. msg.h):

```
struct msg {
   struct msg *msg_next;
   long   msg_type;
   char  *msg_spot;
   short  msg_ts;
};
```

Первый член структуры, `msg_next`, — это указатель на следующее сообщение в очереди. Второй член, `msg_type`, — это тип сообщения, такой же, как в структуре `msg_buf`. Следующий член структуры — это указатель на начало текста сообщения, а последний член, `msg_ts`, размер текста сообщения.

Каждый тип объекта IPC представляется в ядре определенной структурой. Очереди представлены структурой `msqid_ds` (объявлена в файле /usr/src/linux/include/linux/msg.h):

```
struct msqid_ds {
   struct ipc_perm msg_perm;
   struct msg *msg_first;
   struct msg *msg_last;
   time_t msg_stime;
```

```
   time_t msg_rtime;
   time_t msg_ctime;
   struct wait_queue *wwait;
   struct wait_queue *rwait;
   ushort msg_cbytes;
   ushort msg_qnum;
   ushort msg_qbytes;
   ushort msg_lspid;
   ushort msg_lrpid;
};
```

Начнем с первого поля — `msg_perm`, содержащего информацию о владельце и правах доступа. Данная информация записывается в структуру типа `ipc_perm` (определена в файле linux/ipc.h):

```
struct ipc_perm {
   key_t  key;    /* Ключ */
   ushort uid;    /* uid и gid владельца */
   ushort gid;
   ushort cuid;   /* uid и gid создателя */
   ushort cgid;
   ushort mode;   /* Режим доступа */
   ushort seq;    /* Порядковый номер использования гнезда */
};
```

Поле `seq` — системное и вас, как программиста, оно не касается. Каждый раз при уничтожении объекта IPC это поле уменьшается ядром на `MAX`, где `MAX` — максимальное количество объектов IPC в вашей системе.

После поля `msg_perm` следуют два поля, содержащие указатели на первое и последнее сообщение в очереди. Поле `msg_time` содержит время отправки последнего сообщения из очереди. Поле `msg_rtime` содержит время последнего изъятия сообщения из очереди, а `msg_ctime` — время последнего изменения очереди.

Поля `wwait` и `rwait` — указатели на очередь ожидания ядра, которые используются, когда очередь переполнена.

Поле `msg_cbytes` содержит общий объем всех сообщений в очереди, поле `msg_qnum` — количество сообщений в очереди, `msg_qbytes` — максимальный размер очереди.

Последние два поля очень важны: поле `msg_lpid` содержит PID процесса, пославшего последнее сообщение в очередь, а `msg_rpid` — PID процесса, получившего сообщение из очереди.

12.4.3. Создание очереди сообщений

Для создания очереди сообщений используется системный вызов `msgget()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

Первый аргумент — это ключ, который мы получаем с помощью системного вызова `ftok()`. Второй аргумент — это режим доступа к очереди:

- `IPC_CREAT` — создать очередь, если она не была создана ранее;
- `IPC_EXCL` — если использовать вместе с IPC_CREAT, то в случае если очередь существует, мы получим ошибку.

Режим `IPC_CREAT` (без IPC_EXCL) всегда возвращает идентификатор очереди, даже если очередь уже существует (происходит подключение к очереди).

Если использовать `IPC_EXCL` вместе с `IPC_CREAT`, также будет создана новая очередь, но если очередь уже существует, подключения не произойдет, а функция `msgget()` возвратит –1.

Вместе с режимом `IPC_CREAT` можно указывать права доступа к очереди:

```
IPC_CREAT | 0660
```

Если функция `msgget()` вернула –1, проанализировать ошибку можно с помощью переменной `errno`:

- `EACCESS` — у вас нет прав доступа к объекту IPC;
- `EEXIST` — очередь уже существует, создание невозможно, но возможно подключение к очереди;
- `EIDRM` — очередь помечена для удаления;
- `ENOENT` — очередь не существует (в случае подключения);
- `ENOMEM` — не хватает памяти для создания очереди;
- `ENOSPC` — не хватает адресного пространства (т. е. превышено максимальное количество очередей).

Следующий код демонстрирует создание очереди сообщений:

```
key_t key;            /* Ключ IPC */
int id;               /* ID очереди сообщений */

/* Генерируем ключ */
key = ftok ("/etc/passwd", 'd');

/* Создаем очередь */
   if ((id = msgget ( key, IPC_CREAT | 0660 )) == -1)
   {
      printf("Error\n");
      exit(1);
   }
```

12.4.4. Постановка и чтение сообщений

После создания очереди (или подключения к существующей) мы можем с ней работать, т. е. отправлять сообщения в очередь и читать сообщения из очереди. Для постановки сообщения в очередь используется вызов `msgsend()`:

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

Первый аргумент — это идентификатор очереди, в которую нужно добавить сообщение. Данный идентификатор мы предварительно получаем с помощью системного вызова `msgget()`.

Второй параметр — это указатель на буфер сообщения. Третий аргумент — это длина сообщения без учета типа сообщения (4 байта). Последний аргумент обычно устанавливают равным 0 или `IPC_NOWAIT`, если вы не хотите, чтобы процесс был блокирован при постановке сообщения в очередь, например в случае переполнения очереди. По умолчанию (когда флаг равен 0), если очередь переполнена, ваш процесс будет блокирован до тех пор, пока сообщение не будет поставлено в очередь.

Как обычно, в случае успеха вызов возвращает 0, или –1, если произошла ошибка. С помощью `errno` можно анализировать ошибку:

- `EAGAIN` — очередь переполнена, а вы используете флаг `IPC_NOWAIT`, т. е. сообщение будет удалено и вам нужно заново поставить его в очередь;
- `EACCESS` — у вас недостаточно прав для записи сообщения в очередь;
- `EFAULT` — неверный адрес буфера `msgp` (невозможно получить доступ к этому адресу);
- `EIDRM` — очередь сообщений удалена;
- `EINVAL` — ошибка в аргументах;
- `ENOMEM` — не хватает памяти.

Следующий код демонстрирует постановку сообщения в очередь:

```
/* res – результат операции
   length – длина сообщения */
int res, length;
struct my_buf *buf;          /* Буфер сообщения */

/* Вычисляем длину сообщения */
length = sizeof(struct my_buf);

/* Отправляем сообщение в очередь с идентификатором id */
   if((res = msgsnd( id, &buf, length, 0)) == -1)
   {
       printf("Error\n");
       exit(1);
   }
```

Данный пример подразумевает, что очередь уже создана, `id` — идентификатор очереди. Код создания очереди был приведен ранее. Теперь попробуем собрать все

воедино и напишем программу, создающую очередь сообщений и отправляющую в нее сообщение (листинг 12.6).

Листинг 12.6. Программа queue-demo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
   int id;                    /* ID очереди */
   key_t key;          /* Ключ */
   int res, length;   /* Результат операции и длина сообщения */

/* Структура сообщения */
   struct my_buf {
      long   mtype;
/* Произвольные поля очереди */
      int    op_type;         /* Тип операции */
      int    l_op;            /* Первый операнд */
      int    r_op;            /* Второй операнд */
   } msg;

   /* Создаем IPC-ключ */
   key = ftok(".", 'a');

   /* Создаем очередь или присоединяемся к уже существующей */
   if ((id = msgget ( key, IPC_CREAT | 0660 )) == -1)
   {
      printf("Ошибка при создании очереди\n");
      exit(1);
   }

   /* Формируем сообщение */
   msg.mtype    = 1;  /* Тип сообщения, должен быть положительным! */
   msg.op_type = 0;  /* Тип операции */
   msg.l_op = 5;
   msg.r_op = 4;

   /* Отправляем сообщение в очередь */
  length = sizeof(struct my_buf);

   if((res = msgsnd( id, &msg, length, 0)) == -1)
   {
       printf("Ошибка при постановке сообщения в очередь\n");
```

```
        exit(1);
    }

return 0;
}
```

После запуска этой программы запустите программу ipcs. Как видно из рис. 12.1, создана очередь сообщений с правами 0666.

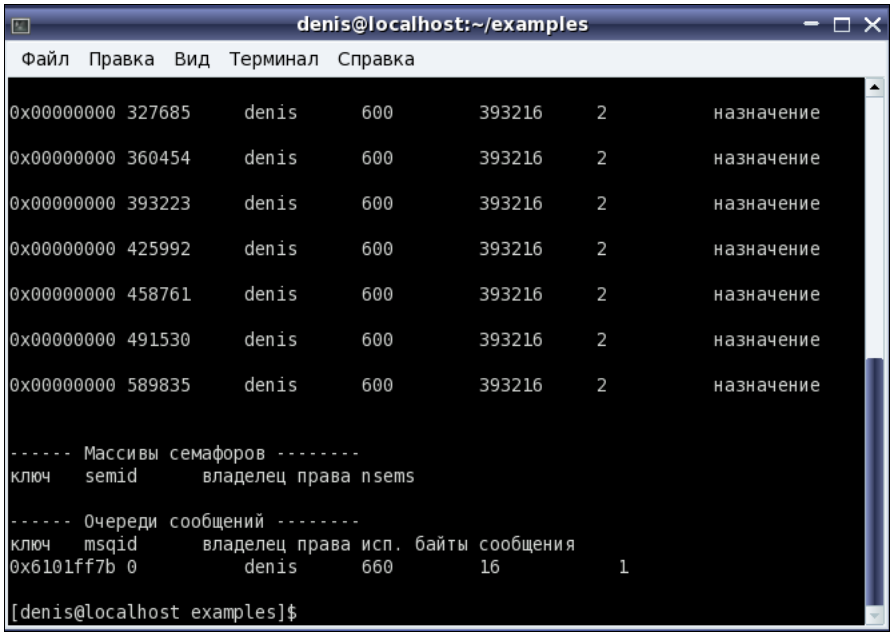

denis@localhost:~/examples
Файл Правка Вид Терминал Справка
0x00000000 327685 denis 600 393216 2 назначение
0x00000000 360454 denis 600 393216 2 назначение
0x00000000 393223 denis 600 393216 2 назначение
0x00000000 425992 denis 600 393216 2 назначение
0x00000000 458761 denis 600 393216 2 назначение
0x00000000 491530 denis 600 393216 2 назначение
0x00000000 589835 denis 600 393216 2 назначение
------ Массивы семафоров --------
ключ semid владелец права nsems
------ Очереди сообщений --------
ключ msqid владелец права исп. байты сообщения
0x6101ff7b 0 denis 660 16 1
[denis@localhost examples]$

Рис. 12.1. Создана очередь сообщений

Теперь напишем программу, которая получит это сообщение. Для получения сообщения используется системный вызов `msgrcv()`:

```
size_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int
msgflg);
```

Первый аргумент определяет очередь, из которой нужно получить сообщение. Второй аргумент — это адрес буфера, в который будет записано сообщение. Третий аргумент — это своеобразный ограничитель длины сообщения. Четвертый аргумент — это тип сообщения. Последний параметр — флаг сообщения. Если вы установили флаг `IPC_NOWAIT`, но в очереди нет ни одного подходящего сообщения, например нет сообщения с нужным вам типом, вашему процессу будет возвращено значение `ENOMSG`.

В случае успеха вызов `msqrcv()` возвращает число байтов, скопированных в буфер, или –1 в случае ошибки. Переменная `errno` устанавливается так:

- `E2BIG` — длина сообщения больше, чем ограничитель `msgsz`;
- `EACCESS` — у вас недостаточно прав;

- EFAULT — недоступен адрес буфера;
- EIDRM — очередь уничтожена ядром; посылается в случае, если приложение ждет подходящего сообщения, а очередь удаляется ядром;
- EINTR — операция прервана поступившим сигналом;
- EINVAL — ошибка в аргументах, например отрицательный размер сообщения или неверный номер очереди;
- ENOMSG — нет сообщения, удовлетворяющего условию. Посылается, если установлен флаг IPC_NOWAIT, в противном случае процесс будет ждать нужного сообщения.

Код для получения сообщения из очереди следующий:

```
int   id;                          /* ID очереди */
int   res, length;           /* Результат операции, длина */
struct my_buf buf;           /* Буфер */
int type=1;                  /* Тип сообщения */

length = sizeof(struct my_buf);

if((res = msgrcv( id, &buf, length, type, 0 )) == -1)
 {
      printf("Error\n");
      exit(1);
 }
```

Если не хочется ждать, пока в очереди появится сообщение нужного типа, то перед самим получением сообщения нужно проверить его наличие в очереди. Для этого напишем функцию msg_exists(), которая будет принимать два параметра: id очереди и type — тип сообщения. В случае успеха функция будет возвращать TRUE, а при ошибке — FALSE.

```
int msg_exists(int id, long type )
{
   int   res;

   if((res = msgrcv( id, NULL, 0, type, IPC_NOWAIT )) == -1)
   {
      return(TRUE);
   }
   else
      return(FALSE);
}
```

Для удаления очереди используется системный вызов msgctl(). Если быть предельно точным, то msgctl() используется для управления очередью, но может использоваться и для удаления:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Первый параметр — ID очереди, второй — команда, третий — буфер, использующийся некоторыми командами этого системного вызова, например `IPC_STAT`, которая записывает в буфер `buf` статус очереди. Для удаления очереди используется команда `IPC_RMID`:

```
msgctl(id, IPC_RMID, NULL);
```

Подробно о вызове `msgctl()` вы можете прочитать в `man 2 msgctl`.

12.5. Семафоры

12.5.1. Введение в семафоры

Семафоры — это средство IPC, управляющее доступом к общим ресурсам, например устройствам. Семафоры не позволяют одному процессу захватить устройство до тех пор, пока с этим устройством работает другой процесс. Семафор может находиться в двух положениях: 0 (устройство занято) и 1 (устройство свободно).

Один семафор практически никогда не используется, на практике обычно используются множества семафоров, например множество семафоров принтеров. Пусть к вашей системе подключено четыре принтера (как подключено — по сети или локально — уже другой вопрос). Есть множество семафоров принтеров, в нем есть четыре семафора — по количеству принтеров. Каждый из семафоров может принимать значение 1 или 0, как уже было сказано. Когда вы отправляете задание на печать без указания принтера, на котором вы бы хотели распечатать задание, с помощью множества семафоров очень просто определить, какой принтер свободен, а какой — занят. Ваше задание будет распечатано на первом свободном принтере. Конечно, данный пример сугубо теоретический, на практике все немного не так.

Семафоры также могут использоваться, как счетчики ресурсов. Представим, что вместо принтера у нас есть какой-то абстрактный контроллер, позволяющий выполнять 100 операций одновременно. Тогда значение семафора было бы равно 100 при условии, что ни одна команда не выполняется. По мере поступления новых заданий менеджер контроллера уменьшал бы значение семафора на 1 для каждой команды, а при выполнении задания увеличивал бы на 1.

12.5.2. Структуры ядра

Для работы с семафорами в ядре есть собственная структура `semid_ds`, определенная в заголовочном файле /usr/src/linux/include/linux/sem.h:

```
struct semid_ds {
   struct ipc_perm sem_perm;   /* Права доступа */
   time_t sem_otime;               /* Время последней операции */
```

```
   time_t sem_ctime;                /* Время последнего изменения */
   struct sem *sem_base;     /* Указатель на первый семафор */
   struct wait_queue *eventn;       /* Очереди ожидания */
   struct wait_queue *eventz;
   struct sem_undo *undo;      /* Запросы undo в этом массиве */
   ushort sem_nsems;                /* Номера семафоров в массиве */
};
```

Данная структура не совсем обычная. В ней есть указатель на первый семафор типа `sem`:

```
struct sem {
   short   sempid;    /* PID последней операции */
   ushort  semval;    /* Текущее значение семафора */
   ushort  semncnt;    /* Число процессов, ожидающих освобожд. рес */
   ushort  semzcnt;   /* Число процессов, ожидающих освоб. всех рес.  */
};
```

Рассмотрим поля этой структуры:

- `sem_pid` — PID процесса, который произвел последнюю операцию с семафором;
- `sem_semval` — текущее значение семафора;
- `sem_semncnt` — число процессов, ожидающих увеличения значения семафора, т. е. освобождения ресурсов;
- `sem_semzcnt` — число процессов, ожидающих освобождения всех ресурсов.

12.5.3. Создание набора семафоров

Сейчас мы создадим множество семафоров. Для этого используется системный вызов `semget()`, этот же вызов используется для подключения к уже существующему множеству семафоров:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget ( key_t key, int nsems, int semflg );
```

Первый параметр — ключ IPC, который, как и в случае с очередью, можно получить системным вызовом `ftok()`. Второй параметр — это количество семафоров в множестве, третий параметр — флаги множества семафоров. Как и в случае с очередью, вы можете задать флаги `IPC_CREAT` и `IPC_EXCL`. При создании множества семафоров можно также указать права доступа:

```
IPC_CREAT | 0660
```

Количество семафоров в наборе ограничено в файле sem.h:

```
#define SEMMSL 32 /* <= 512 */
```

Второй параметр игнорируется, если производится подключение к уже существующему множеству, а не создается новое.

Функция semget() возвращает идентификатор семафора или –1 в случае ошибки. Переменная errno устанавливается следующим образом:

- EACCESS — у вас не хватает полномочий для выполнения операции;
- EEXISTS — множество существует, его нельзя создать;
- EIDRM — множество помечено для удаления;
- ENOENT — множество не существует, не было ни одной операции IPC_CREAT;
- ENOMEM — не хватает памяти;
- ENOSPC — достигнуто максимальное количество семафоров.

Пример использования вызова semget():

```
if((sid = semget(key, n, IPC_CREAT|IPC_EXCL|0666)) == -1)
 {
    printf("Множество уже существует\n");
    exit(1);
 }
```

12.5.4. Операции над семафорами

Системный вызов semop() используется для операций над множеством семафора:

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Первый аргумент — это идентификатор семафора, возвращаемый вызовом semget(). Второй — это массив операций, который нужно выполнить над семафорами. Последний аргумент — это количество операций в массиве.

Второй аргумент — это массив структур типа sembuf:

```
struct sembuf {
   ushort sem_num;   /* Номер семафора в массиве */
   short  sem_op;    /* Операция над семафором */
   short  sem_flg;   /* Флаги */
};
```

Поле sem_num — номер семафора, над которым нужно выполнить операцию. Второе поле (sem_op) — выполняемая операция. Третье поле — флаги операции, например IPC_NOWAIT. Если IPC_NOWAIT не установлен, то процесс "заснет" до тех пор, пока не освободится указанное количество ресурсов (пока другой процесс не освободит их).

Чтобы понять, как работает semop(), вернемся к примеру о контроллере. Пусть наш контроллер может выполнять 100 операций. Когда мы помещаем команду в контроллер, мы "забираем" один ресурс контроллера, структура sembuf будет заполнена так:

```
struct sembuf dev_lock = { 0, -1, IPC_NOWAIT };
semop(sid, &dev_unlock, 1);
```

Когда команда выполнена, нам нужно "вернуть" ресурс, следовательно, структура `sembuf` будет заполнена так:

```
struct sembuf dev_unlock = { 0, 1, IPC_NOWAIT };
semop(sid, &dev_unlock, 1);
```

В случае успеха, когда выполнены все операции, системный вызов `semop()` возвращает 0. В случае ошибки возвращается –1, а переменная `errno` равна:

- `E2BIG` — количество операций (аргумент `nsops`) превышает разрешенное число операций;
- `EACCESS` — не хватает полномочий;
- `EAGAIN` — операция не может быть выполнена (при использовании флага `IPC_NOWAIT`), такую операцию нужно повторить снова;
- `EFAULT` — указатель `sops` указывает на ошибочный адрес;
- `EIDRM` — множество помечено на удаление;
- `EINTR` — прервано сигналом;
- `EINVAL` — неверный `semid`;
- `ENOMEM` — не хватает памяти для создания структуры Undo-операции;
- `ERANGE` — значение семафора вышло за пределы допустимых значений.

12.5.5. Управление семафором

Для управления семафором используется системный вызов `semctl()`:

```
int semctl ( int semid, int semnum, int cmd, union semun arg );
```

Первый аргумент задает идентификатор семафора, второй — номер семафора во множестве семафоров (нумерация начинается с 0). Третий аргумент — это команда, которую нужно выполнить над семафором. Последний аргумент — это объединение (union) аргументов, которые можно использовать для управления семафором.

Команды, т. е. возможные значения третьего аргумента, представлены в табл. 12.1.

Таблица 12.1. *Команды управления семафором*

Команда	Описание
`IPC_STAT`	Запоминает структуру `semid_ds` для множества по адресу `buf` объединения `semun` (см. далее)
`IPC_SET`	Устанавливает значение члена `ipc_perm` структуры `semid_ds`
`IPC_RMID`	Удаляет множество семафоров
`GETALL`	Позволяет получить значения всех семафоров. Значения возвращаются в виде массива unsigned short, на который указывает член объединения `array`
`GETNCNT`	Возвращает число процессов, которые ожидают ресурсы в данный момент

Таблица 12.1 *(окончание)*

Команда	Описание
GETPID	Возвращает PID процесса, выполнившего последний вызов semop()
GETVAL	Возвращает значение одного семафора
GETZCNT	Возвращает число процессов, которые ожидают полного освобождения ресурса
SETALL	Устанавливает значение семафоров. Значения берутся из члена array объединения semun
SETVAL	Устанавливает значение конкретного семафора. Значение берется из элемента val объединения semun

Теперь рассмотрим объединение аргументов, т. е. последний аргумент функции semctl():

```
union semun {
   int val;                /* Значение для SETVAL */
   struct semid_ds *buf;   /* Буфер для IPC_STAT и IPC_SET */
   ushort *array;          /* Массив для GETALL и SETALL */
   struct seminfo *__buf;  /* Буфер для IPC_INFO */
   void *__pad;
};
```

В случае успеха системный вызов semctl() возвращает натуральное число, а в случае ошибки — –1. Переменная errno равна:

- EACCESS — не хватает полномочий;
- EFAULT — адрес arg ошибочен;
- EIDRM — множество помечено для удаления;
- EINVAL — неправильный аргумент semid;
- EPERM — у вас нет прав для выполнения команды cmd;
- ERANGE — значение семафора вышло за пределы допустимых значений.

Рассмотрим несколько примеров. Пусть у нас есть множество семафоров с идентификатором id, нам нужно получить значение семафора с номером n. Мы будем использовать команду GETVAL (см. табл. 12.1):

```
int value;
int N;
value=semctl(id, N, GETVAL, 0);
```

Теперь рассмотрим получение первых пяти семафоров:

```
int c, val;
for (c = 0; x < 5; c++)
```

```
{
  val=semctl(sid,c,GETVAL,0);
  printf("Семафор %d: %d\n",c,val);
}
```

12.6. Разделяемые сегменты памяти

12.6.1. Структуры ядра

Наиболее быстрым способом IPC являются разделяемые сегменты памяти. Идея очень проста: мы записываем значение в "общий" сегмент памяти, а другие процессы могут получить к нему доступ. В итоге нам не нужны ни каналы, ни очереди сообщений, все гораздо проще.

Для каждого разделяемого сегмента памяти ядро поддерживает специальную структуру — `shmid_ds`, описанную в файле /usr/src/linux/include/linux/shm.h:

```
   struct shmid_ds {
      struct ipc_perm shm_perm;         /* Права доступа */
      int    shm_segsz;                 /* Размеры сегмента в байтах */
      time_t shm_atime;                 /* Время последней привязки */
      time_t shm_dtime;                 /* Время последней отвязки */
      time_t shm_ctime;                 /* Время последнего изменения */
      unsigned short shm_cpid;          /* PID создателя */
       /* PID последнего процесса-пользователя сегмента */
      unsigned short shm_lpid;
      short  shm_nattch;                /* Число привязок */
      unsigned short shm_npages;        /* Размеры сегмента (в страницах) */
      /* Массив указателей на $frames -> S$*/
      unsigned long *shm_pages;
         struct vm_area_struct *attaches;  /* Дескрипторы для привязок */
};
```

На самом деле структура немного сокращена, в ней представлены только самые нужные поля. Полную версию структуры вы найдете в shm.h.

Перед тем как рассмотреть системный вызов `shmget()`, нужно разобраться, что такое *привязка*. Привязка — это размещение сегмента в адресном пространстве процесса, подключение к разделяемому сегменту памяти (далее просто сегменту памяти). Отвязка — отключение. Поле `shm_nattach` содержит количество привязок к сегменту памяти на данный момент.

12.6.2. Создание разделяемого сегмента памяти и привязка к нему

Для создания нового сегмента памяти используется системный вызов `shmget()`. Этот же вызов используется для подключения к уже существующему сегменту:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

Назначение параметров, думаю, понятно с первого взгляда. Первый параметр — это ключ, полученный с помощью функции `ftok()`. Второй — размер сегмента памяти в байтах, а третий — флаги. Если установлен флаг `IPC_CREAT`, системный вызов создаст новый разделяемый сегмент или подключится к уже существующему сегменту, если обнаружится, что такой сегмент уже существует. Если установлен флаг `IPC_EXCL` вместе с `IPC_CREAT`, подключение к существующему сегменту запрещается. Сам по себе флаг `IPC_EXCL` бесполезен.

Системный вызов `shmget()` возвращает идентификатор разделяемого сегмента или –1, если произошла ошибка. Переменная `errno` в случае ошибки устанавливается так:

- `EACCESS` — нет доступа к сегменту;
- `EINVAL` — неправильно заданы размеры сегмента;
- `EEXISTS` — сегмент уже существует, создание невозможно. Вы получите эту ошибку, если будете использовать флаг `IPC_EXCL` вместе с `IPC_CREAT` при условии, что сегмент уже существует;
- `IDRM` — сегмент помечен на удаление или уже удален;
- `ENOMEM` — не хватает памяти для создания сегмента.

Создадим разделяемый сегмент памяти:

```
shid = shmget( key, size, IPC_CREAT | 0660 )
```

После получения идентификатора сегмента вы должны привязаться к нему, для чего используется системный вызов `shmat()`:

```
int shmat ( int shmid, char *shmaddr, int shmflg );
```

Первый аргумент — это идентификатор сегмента, который мы получаем с помощью предыдущего вызова. Второй аргумент — это адрес привязки. Рекомендуется указать 0 вместо адреса: ядро само найдет нераспределенную область.

Третий аргумент — это флаги. Обычно используется два флага:

- `SHM_RND` — переданный адрес будет округлен до ближайшей страницы (если вы сами указываете адрес);
- `SHM_RDONLY` — сегмент памяти будет доступен только для чтения.

В случае успеха `shmat()` возвращает адрес, по которому сегмент был привязан к процессу, или –1, если произошла ошибка. Переменная `errno` может принимать всего три значения:

- `EACCESS` — нет доступа;
- `ENOMEM` — не хватает памяти;
- `EINVAL` — ошибка в параметрах, т. е. неправильное значение ID или адреса привязки (`shmaddr`).

Пример привязки:

```
char *ptr;
ptr = shmat(sh_id,0,0);
```

После привязки сегмента к адресному пространству доступны операции чтения и записи, которые очень напоминают работу с простыми указателями. Чуть позже мы рассмотрим программу для демонстрации работы с разделяемыми сегментами памяти, но прежде нам нужно знать, как снять привязку и как управлять сегментом.

Для снятия привязки используется системный вызов `shmdt()`:

```
int shmdt ( char *shmaddr );
```

В случае ошибки данный системный вызов возвращает –1. Значение `errno` только одно: `EINVAL`, т. е. вы неправильно указали адрес привязки.

Для управления РСП используется системный вызов `shmctl()`:

```
int shmctl ( int shmid, int cmd, struct shmid_ds *buf );
```

Первый параметр — идентификатор сегмента, второй — команда, а третий — буфер, необходимый для команд `IPC_STAT`/`IPC_SET`.

Команд всего три:

- `IPC_STAT` — сохраняет структуру `shmid_ds` по адресу `buf`;
- `IPC_SET` — берет значение элемента `ipc_perm` структуры `shmid_ds` и устанавливает его для сегмента. Значение берется из `buf`;
- `IPC_RMID` — помечает сегмент для удаления, само удаление произойдет как только последний процесс отвяжется от сегмента. Если сегмент помечен на удаление, ни один процесс не сможет привязаться к сегменту.

Как обычно, в случае успеха системный вызов `shmctl()` возвращает 0 или –1, если произошла ошибка.

12.6.3. Демонстрационная программа

Как я и обещал, рассмотрим программу, демонстрирующую работу с разделяемыми сегментами памяти (листинг 12.7).

Листинг 12.7. Программа share.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

/* Размер нашего сегмента – 256 байтов */
#define SIZE 256
```

```
int main(int argc, char *argv[])
{
        key_t key;                    /* Ключ */
        int shmid, c;          /* Идентификатор сегмента */
        char *ptr;                    /* Указатель */

        /* Генерируем ключ */
        key = ftok(".", 'D');

        /* Создаем сегмент или подключаемся к уже существующему */
        shmid = shmget(key, SIZE, IPC_CREAT|0660);

        /* Привязываемся к сегменту */
        if((ptr = shmat(shmid, 0, 0)) == -1)
        {
              printf("shmat() error\n");
              exit(1);
        }

        /* Далее можно работать с ptr как с обычным указателем */

        return 0;
}
```

После запуска программы запустите программу ipcs и убедитесь, что наша программа создала разделяемый сегмент памяти.

Напоследок рассмотрим функцию изменения прав доступа к сегменту. Первый параметр этой функции — идентификатор сегмента, второй — новые права доступа:

```
shm_change_mode(int shmid, char *mode)
{
   struct shmid_ds mds;

   shmctl(shmid, IPC_STAT, &mds);

   printf("Старые права доступа: %o\n", mds.shm_perm.mode);

   sscanf(mode, "%o", &mds.shm_perm.mode);

   shmctl(shmid, IPC_SET, &mds);

   printf("Новые права доступа: %o\n", mds.shm_perm.mode);
}
```

Глава получилась довольно большой и сложной, но расслабляться не приходится: следующая глава посвящена созданию модулей ядра.

ГЛАВА 13

Создание модуля ядра

13.1. Что такое модуль ядра

Что же такое ядро, святая святых Linux? Можно сказать, что ядро — это основная программа операционной системы. Именно ядро отвечает за поддержку устройств, файловых систем, сетевых протоколов. Сейчас мы не будем рассматривать все возможности ядра — об этом можно написать целую книгу, — а остановимся только на модулях ядра.

Представим, что мы сейчас работаем в Windows и добавили новое устройство. Чтобы оно заработало, нужно установить драйвер. Где его взять (скачать с Интернета, установить с диска или использовать стандартный драйвер, если это возможно) — уже другой вопрос. Но нам нужна программа, которая "научит" нашу систему "общаться" с новым устройством, — нам нужен драйвер.

В Linux для поддержки какого-нибудь устройства нужно подключить модуль ядра. Нет, не следует думать, что модуль ядра — это драйвер. Некоторые пользователи называют модули ядра драйверами, но это не совсем так. Да, модуль ядра может выполнять функции, аналогичные драйверу в Windows, но это частный случай.

Как уже было отмечено, ядро отвечает за поддержку файловых систем, устройств, сетевых протоколов и этот список можно продолжить, но пока мы остановимся на этих трех пунктах. Чтобы ваша система могла использовать, скажем, сетевую файловую систему NTFS или поддерживать SCSI-диски, в ядре должна быть поддержка NTFS и SCSI-устройств. Чтобы включить ту или иную опцию (поддержка NTFS, поддержка SCSI), нужно перекомпилировать ядро. В программе конфигурации вы определяете, какие возможности вам нужны, а какие — нет.

Но вы только представьте, что произойдет с ядром, если в него включить поддержку *всех* устройств, *всех* файловых систем, *всех* протоколов и включить *все* прочие опции, которые теоретически могут понадобиться пользователю? Правильно, ядро станет неприлично большим и не сможет загружаться на слабых компьютерах, а вся система будет ужасно "тормозить" — ведь у нее такое нерасторопное ядро.

Выход из ситуации — включать только самые необходимые опции, которые могут понадобиться на этапе загрузки системы. Например, если загрузка производится со

SCSI-винчестера, в состав ядра должна быть включена поддержка SCSI. Также должна быть включена поддержка файловых систем ext2, ext3, ext4 — "родных" файловых систем Linux.

А как же быть с остальными опциями ядра, которые могут понадобиться пользователю? Перекомпилировать ядро — не выход, учитывая, сколько времени занимает эта операция. Для решения этой задачи и были созданы модули. Модуль позволяет расширить функциональность ядра без его перекомпиляции и даже без перезагрузки системы.

Предположим, что вы скомпилировали ядро с поддержкой SCSI и ext3. Вы загрузили систему, и вам нужно прочитать NTFS-диск. Вы с помощью команды `insmod` внедряете в ядро соответствующий модуль, и у вас появляется поддержка ядра — даже не нужно перезагружать систему. Если модуль нужен постоянно, то он прописывается в /etc/modules.conf — при загрузке ядро читает этот файл и загружает указанные в нем модули.

Модуль может добавить в ядро любой код — обеспечить поддержку какого-нибудь устройства, файловой системы, сетевого протокола и т. д. Получается, что с помощью модулей вы без перекомпиляции ядра можете легко и, главное, быстро расширить функциональность вашей системы.

Когда-то очень давно я рекомендовал перекомпилировать ядро, удалять из него поддержку всего, что вам не нужно, но включать только необходимые опции. Поддержку модулей рекомендовал отключить из соображений безопасности, а опции, предлагаемые в виде модулей, включить сразу в состав ядра (если они были нужны пользователю). Такое ядро называется *монолитным*. Сейчас на практике подобное ядро вы уже не встретите — разве что на встроенных системах с ограниченной функциональностью. Раньше размер ядра был немного другим, да и компьютеры были несколько иными — более простыми, даже если включить все опции, необходимые пользователю, размер ядра получался всего несколько мегабайтов. Сейчас я даже не могу представить, сколько будет "весить" монолитное ядро для домашнего компьютера.

Однако все это — уже ликбез и мало кому интересный разговор. Самое важное — что такое модуль — мы уже выяснили, поэтому самое время перейти к следующему разделу.

> ***Примечание***
>
> Программирование модулей ядра в Linux, как уже отмечалось — тема для отдельной книги, поэтому глупо было бы полагать, что в этой книге вы найдете исчерпывающие сведения.

13.2. Команды *lsmod*, *insmod*, *modprobe*

Разберемся, как модули попадают в ядро. Один из способов — это использование команды `insmod` вручную. Вы просто вводите эту команду, а в качестве аргумента указываете имя модуля, который вы хотите добавить в ядро, например:

```
# insmod /путь/модуль.o
```

Второй способ — это использование файла /etc/modules.conf (в некоторых дистрибутивах — просто /etc/modules). Вы прописываете в этом файле все модули ядра, необходимые вам постоянно, дабы не вводить команду `insmod` после каждой перезагрузки для каждого модуля.

В более современных дистрибутивах вместо файла /etc/modules.conf используется файл /etc/modprobe.conf, формат которого такой же, как у modules.conf. В листинге 13.1 приведен пример файла /etc/modprobe.conf.

Листинг 13.1. Файл /etc/modprobe.conf

```
alias eth0 pcnet32
alias sound-slot-0 snd_ens1371
install scsi_hostadapter /sbin/modprobe mptscsih; /sbin/modprobe mptspi;
/sbin/modprobe ata_piix; /sbin/modprobe ahci; /bin/true
install usb-interface /sbin/modprobe ehci_hcd; /sbin/modprobe uhci_hcd;
/bin/true
```

Получить подробную информацию о формате этого файла можно в справочной системе:

```
man modprobe.conf
```

Команда `modprobe`, как правило, вызывается при загрузке системы из сценариев инициализации системы. Далее она читает свой файл конфигурации и загружает указанные в нем модули (или, наоборот, выгружает, если указана команда `remove`).

Команду `modprobe` можно использовать и для загрузки модулей. При этом ее удобнее использовать, даже чем `insmod`. Команда `insmod` ничего не знает о размещении модулей, и вам для загрузки модуля нужно указывать полный путь к нему вместе с "расширением" файла модуля. Команду `modprobe` использовать проще — нужно указать только имя модуля (без пути к файлу), например:

```
# modprobe ivtv
```

Данная команда найдет и загрузит модуль `ivtv`. Как видите, вам даже не нужно знать, в каком файле находится файл модуля.

Чуть ранее было сказано, что вместо файла /etc/modules.conf используется файл /etc/modprobe.conf. Это не совсем так. Модули, указанные в файле /etc/modules.conf (или просто в /etc/modules), загружаются при загрузке системы. А вот модули, указанные в /etc/modprobe.conf, — при необходимости, например, когда ядру понадобилась поддержка того или иного устройства. В системе очень много модулей, но это не означает, что все эти модули нужно перечислять в /etc/modprobe.conf. При необходимости (даже если модуль не указан в /etc/modprobe.conf) программа `modprobe` и так найдет и загрузит его. Файл /etc/modprobe.conf используется для передачи параметров модулям ядра — иными словами, для конфигурации этих модулей. Например, вы можете передать модулю звуковой карты начальный уровень громкости. Конкретный пример приводить не стану, т. к. все зависит от реализации самого модуля.

Что же касается файла /etc/modules.conf, то в современных дистрибутивах действительно есть его аналог — /etc/modprobe.preload. В нем указываются модули, которые должны быть загружены при старте системы.

Кроме файлов /etc/modprobe.conf и /etc/modprobe.preload обычно имеются каталоги /etc/modprobe.d и /etc/modprobe.preload.d. Зайдите в один из этих каталогов. В нем вы найдете несколько текстовых файлов, в каждом из которых — команды по управлению модулями. Все эти команды можно было бы прописать в файле /etc/modprobe.conf или в /etc/modprobe.preload. Но для большего удобства, особенно если команд много, можно их вынести в отдельный файл и поместить его в каталог /etc/modprobe.d (или в /etc/modprobe.preload.d, если модуль нужно загружать при старте системы).

Есть еще один способ — загрузка модулей из каталога /etc/sysconfig/modules. Вообще способы загрузки модулей зависят от дистрибутива — возможно, в вашем дистрибутиве будет еще один способ, о котором в этой книге ничего не сказано. Например, в Ubuntu нет файла /etc/modprobe.conf, но есть файл /etc/modules и каталог /etc/modprobe.d. Сейчас мы не будем рассматривать все дистрибутивы подряд, поскольку наша задача — написать модуль ядра, а уж как загрузить его, вы наверняка знаете — вряд ли неопытный Linux-пользователь возьмется за написание модуля ядра.

Итак, в каком же файле лучше прописывать модули? Учитывая, что время не стоит на месте, то лучше все-таки их указывать либо в /etc/modprobe.conf, либо в /etc/modprobe.preload (или создать отдельный файл в соответствующем каталоге). Файлы /etc/modules.conf и /etc/modules могут даже не обрабатываться в некоторых дистрибутивах. Но в файловой системе будет один из этих файлов для обеспечения обратной совместимости.

В любом случае разработанные нами модули вы будете загружать с помощью команды `insmod` — она работает везде, а потом уже разберетесь с автоматической загрузкой модулей в вашем дистрибутиве. Команда `lsmod` позволяет вывести список загруженных модулей. Данную команду нужно вводить от имени root и желательно перенаправить вывод на программу `less` — уж слишком много модулей будет в выводе `lsmod`:

```
# lsmod | less
```

Удалить загруженный модуль из ядра можно командой `rmmod`. Понятно, что команда удаляет модуль из ядра, но файл модуля никуда не денется с жесткого диска.

13.3. Установка необходимых пакетов

Прежде чем приступить к разработке собственного модуля, вам нужно установить дополнительные пакеты:

- make — содержит утилиту `make`, без которой просто невозможно выполнить сборку модуля ядра (точнее, возможно, но очень и очень неудобно), хотя, скорее всего, эта утилита уже у вас установлена;

- kernel-source — содержит исходные тексты ядра, пакет kernel-source называется одинаково во всех дистрибутивах;
- kernel-headers, или linux-userspace-headers — заголовочные файлы ядра, название пакета может отличаться в зависимости от дистрибутива, произведите поиск по строке headers и прочитайте описание найденных пакетов — найти нужный не составит труда;
- kernel-default-devel — файлы, необходимые для разработки модулей ядра. В некоторых дистрибутивах этот пакет может называться иначе, например kernel-devel или kernel-*-devel.

Остальные необходимые пакеты, как правило, будут установлены автоматически при разрешении зависимостей.

13.4. Ваш первый модуль

Наша задача — написать и скомпилировать ваш первый модуль, точнее — болванку модуля. Наш модуль не будет выполнять каких-либо полезных действий, но вы сможете его загрузить в ядро и выгрузить из ядра командой `rmmod`.

Написание "болванки" начнем с подключения заголовочного файла module.h, необходимого для каждого модуля:

```
#include <linux/module.h>
```

Также желательно подключить файл kernel.h — в нем содержатся полезные константы:

```
#include <linux/kernel.h>
```

Если вы хотите просмотреть подключаемые файлы, то их стоит искать не в каталоге /usr/include/linux, а в /usr/src/linux/include/linux.

Любой модуль содержит функцию `init_module()`, которая запускается при загрузке модуля в ядро. При выгрузке модуля из ядра вызывается другая функция — `cleanup_module()`. Наш простой модуль будет состоять из этих двух функций. Для демонстрации компиляции и использования модуля больше ничего не нужно. Но, чтобы создать видимость работы модуля, в эти функции мы добавим вызов функции `printk()`. Функция `printk()` позволяет отправить сообщение в системный журнал, обычно это /var/log/messages. Впрочем, об этой функции мы поговорим отдельно.

Итак, наших знаний достаточно, чтобы написать "болванку" модуля (листинг 13.2).

Листинг 13.2. Модуль first.c

```
#include <linux/module.h>
#include <linux/kernel.h>


int init_module(void)
```

```
{
        printk(KERN_ALERT "Hello from kernel!\n");
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_ALERT "first.c: removed\n");
}
```

Нужно отметить, что функция `init_module()` должна возвратить 0, что свидетельствует о том, что ошибок нет. Если возвратить ненулевое значение, модуль загружен не будет, т. к. будет считаться, что при загрузке произошла ошибка. Функция `cleanup_module()` может не возвращать никаких значений.

Теперь рассмотрим функцию `printk()`. Это очень важная функция, она позволяет отправить сообщение в системный журнал (конечно, при условии, что запущен *демон протоколирования* — syslog, klogd, syslog-ng или любой другой).

Сама функция `printk()` не производит запись в системный журнал. Она записывает ваше сообщение в специальный буфер ядра, который и читается демоном протоколирования. Обычно демон klogd читает сообщения из этого буфера и передает демону syslogd, а тот, в свою очередь, записывает их в системный журнал.

Перед самим сообщением я указал уровень протоколирования сообщения — `KERN_ALERT`. Это очень высокий уровень протоколирования, но благодаря этому сообщение будет не только передано демону протоколирования, но и выведено на консоль. Вообще, выводить сообщения на консоль нужно только в самых критических ситуациях, старайтесь избегать вывода на консоль просто так — это считается дурным тоном. Но при отладке своего модуля вы можете выводить сообщение на консоль, чтобы каждый раз не просматривать системный журнал.

Уровни протоколирования сообщения приведены в табл. 13.1.

Таблица 13.1. *Уровни протоколирования сообщений*

Уровень	Константа	Описание
7	`KERN_DEBUG`	Отладочные сообщения, самый низкий приоритет
6	`KERN_INFO`	Информационные сообщения
5	`KERN_NOTICE`	Это уже не информационное сообщение, но еще и не предупреждение
4	`KERN_WARNING`	Предупреждение: скоро может пойти что-то не так
3	`KERN_ERR`	Возникла ошибка
2	`KERN_CRIT`	Возникла критическая ошибка
1	`KERN_ALERT`	Тревога — система скоро "развалится"
0	`KERN_EMERG`	Система "развалилась" (система больше не может использоваться)

Сообщения с уровнем KERN_ERR и ниже (KERN_CRIT, KERN_ALERT, KERN_EMERG) обычно выводятся на консоль, хотя это зависит от настройки системы, но все сообщения записываются в системный журнал при условии, что демон протоколирования запущен и сама запись возможна. Например, KERN_EMERG может свидетельствовать об отказе жесткого диска, поэтому записать сообщение будет невозможно.

Конечно, если ничего страшного не случилось, а вы просто вывели сообщение с уровнем KERL_ALERT, на "здоровье" системы это никак не отразится.

Начиная с ядра версии 2.3.13 вы можете отказаться от стандартных названий функций init_module() и cleanup_module(). Вместо этих названий вы можете использовать собственные. Однако вам нужно указать, как называются новые функции инициализации и деинициализации модуля. Для этого используются макросы module_init() и module_exit(), определенные в linux/init.h. В листинге 13.3 мы переопределили имена стандартных функций.

Листинг 13.3. Модуль first2.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init first_init(void)
{
        printk(KERN_ALERT "Hello from kernel\n");
        return 0;
}

static void __exit first_exit(void)
{
        printk(KERN_ALERT "first.c: removed\n");
}

module_init(first_init);
module_exit(first_exit);
```

В заголовочном файле module.h находятся также макросы, позволяющие указать информацию о модуле:

```
MODULE_LICENSE("GPL");                              /* Лицензия */
MODULE_AUTHOR("Denis Kolisnichenko");       /* Автор */
MODULE_DESCRIPTION("Driver for /dev/mydev"); /* Описание */
MODULE_SUPPORTED_DEVICE("mydev");           /* Поддерживаемое устройство */
```

Все эти макросы на данном этапе не нужны, но понадобятся, когда вы будете создавать реальный модуль.

13.5. Компиляция модуля

Компиляция модуля — это процесс, где можно легко наткнуться на множество подводных камней. Вроде бы все делаешь правильно, но модуль не собирается. Вроде бы установил все заголовочные файлы, но компилятор упорно сообщает, что файл linux/module.h отсутствует, хотя вы уже десять раз убедились, что это не так...

Чтобы все работало, как следует, вам первым делом нужно сделать две вещи:

- убедиться, что установлены исходные коды ядра, а также заголовочные файлы ядра (см. список пакетов, приводившийся ранее);
- отказаться от использования X.Org (X Window).

Чем же не угодил X.Org? Дело в том, что сообщения, которые будут выводить ваши модули с помощью `printk()`, не будут отображаться в окне терминала! Они будут занесены в системный журнал, но в окне терминала вы их не увидите. Модули не могут выводить сообщения на экран с помощью обычной функции `printf()` — это особенность модулей. Поэтому либо вы с комфортом работаете в консоли и видите все отладочные сообщения, выводимые модулем с помощью `printk()`, либо вы работаете в X.Org и читаете системный журнал /var/log/messages.

Я работаю с Linux уже 12 лет (с 1999 года). Во времена ядер 2.2 и 2.4 модули собирались немного иначе. Вполне может сложиться ситуация, что вы нашли какое-то старое руководство и пытаетесь читать его одновременно с этой книгой (или же читали до чтения этой книги). Раньше для компиляции модуля нужно было ввести команду:

```
gcc -DLINUX -DMODULE -D__KERNEL__ -c имя_файла.c
```

Сейчас, если вы введете такую команду, вы получите сообщение о том, что файл linux/module.h не найден. Даже если вы точно укажете путь (с помощью опции `-I`), то компилятор `gcc` не найдет какой-то другой заголовочный файл — у меня отказались находиться файлы asm/*.h. Разбираться с этой проблемой в системе с новым ядром я не стал — нет смысла, поскольку теперь процесс сборки модуля ядра полностью автоматизирован. С появлением `kbuild` процесс сборки посторонних модулей теперь полностью интегрирован в механизм сборки ядра. Компилировать модули — сплошное удовольствие.

Итак, начнем. Пусть наш модуль first.c находится в каталоге /root/module. Создайте в этом каталоге файл Makefile следующего содержания:

```
obj-m := first.o
```

Да, это и есть наш Makefile. Всего одна строчка. Раньше Makefile для сборки одного модуля состоял как минимум из пяти строчек.

Теперь, находясь в каталоге /root/module, введите команду:

```
# make -C /usr/src/linux SUBDIRS=$PWD modules
```

Разберемся, что означает эта команда. Как уже отмечалось, параметр `-C` задает каталог, в котором следует искать Makefile. Как видите, мы указываем не наш

Makefile, а Makefile ядра. Обычно /usr/src/linux — это символическая ссылка на каталог /usr/src/linux-2.6.x.x. Если в вашей системе нет такой ссылки, то или создайте ее, или укажите в команде путь к исходным текстам ядра.

Команда `make` должна собрать цель modules (ведь мы же компилируем модуль!) из файла /usr/src/linux/Makefile (в нашем Makefile такой цели нет). А файлы, которые нужно скомпилировать, программа должна искать в текущем каталоге (значение переменной `$PWD` — именно поэтому важно вводить команду `make` из каталога, содержащего модуль). В этом каталоге программа `make` найдет наш Makefile модуля и обработает его.

Результат компиляции модуля изображен на рис. 13.1. Компилирование нашего простейшего модуля займет всего пару секунд. Но это только в том случае, если все пройдет гладко. Мы сначала проанализируем вывод команды `make`, а затем рассмотрим нештатные ситуации. Из рис. 13.1 видно, что у нашего модуля отсутствует информация о зависимостях и описание версии модуля. Этим мы займемся позже — у нашего модуля пока не может быть никаких зависимостей, так что на предупреждение можете не обращать внимания. Начиная с ядра 2.6 файлы модулей имеют "расширение" ko вместо o, чтобы их можно было легко отличить от обычных объектных файлов. В нашем случае создан модуль /root/module/first.ko.

```
[root@localhost module]# make -C /usr/src/linux SUBDIRS=$PWD modules
make: Entering directory `/usr/src/linux-2.6.33.5-2mnb'

  WARNING: Symbol version dump /usr/src/linux-2.6.33.5-2mnb/Module.symvers
           is missing; modules will have no dependencies and modversions.

  CC [M]  /root/module/first.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/module/first.mod.o
  LD [M]  /root/module/first.ko
make: Leaving directory `/usr/src/linux-2.6.33.5-2mnb'
[root@localhost module]#
```

Рис. 13.1. Модуль скомпилирован

```
***
*** You have not yet configured your kernel!
*** (missing kernel config file ".config")
***
*** Please run some configurator (e.g. "make oldconfig" or
*** "make menuconfig" or "make xconfig").
***
make[2]: *** [silentoldconfig] Ошибка 1
make[1]: *** [silentoldconfig] Ошибка 2

The present kernel configuration has modules disabled.
Type 'make config' and enable loadable module support.
Then build a kernel with module support enabled.

make: *** [modules] Ошибка 1
```

Рис. 13.2. Отсутствует конфигурация ядра

Теперь поговорим о нештатных ситуациях. При первой сборке модуля программа `make` может сообщить, что отсутствует конфигурация ядра (рис. 13.2). Во всяком случае, такую ошибку я получил при первой сборке своего модуля в openSUSE 11.2 и Mandriva 2010.1 — в других дистрибутивах я не пробовал собирать демонстрационный модуль.

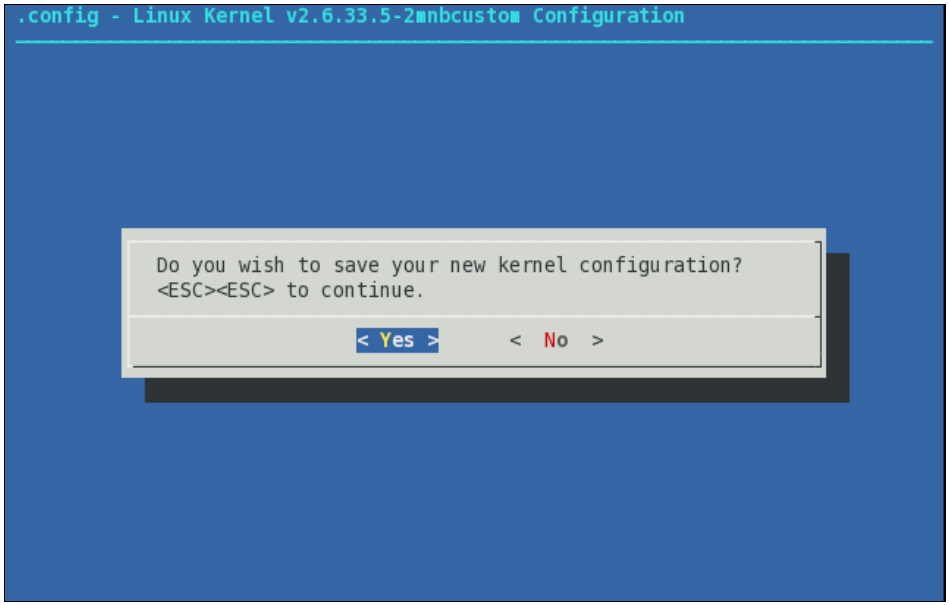


Рис. 13.3. Сохранить конфигурацию ядра?

```
[root@localhost module]# cd /usr/src/linux
[root@localhost linux]# make menuconfig
  HOSTCC  scripts/kconfig/lxdialog/checklist.o
  HOSTCC  scripts/kconfig/lxdialog/inputbox.o
  HOSTCC  scripts/kconfig/lxdialog/menubox.o
  HOSTCC  scripts/kconfig/lxdialog/textbox.o
  HOSTCC  scripts/kconfig/lxdialog/util.o
  HOSTCC  scripts/kconfig/lxdialog/yesno.o
  HOSTCC  scripts/kconfig/mconf.o
  HOSTLD  scripts/kconfig/mconf
scripts/kconfig/mconf arch/x86/Kconfig
#
# using defaults found in /boot/config-2.6.33.5-server-2mnb
#
#
# configuration written to .config
#


*** End of Linux kernel configuration.
*** Execute 'make' to build the kernel or try 'make help'.
```

Рис. 13.4. Конфигурация ядра сохранена

Избавиться от проблемы очень просто. Перейдите в каталог /usr/src/linux и введите команду:

```
make menuconfig
```

В появившемся окне нажмите кнопку **Exit**. После чего `menuconfig` спросит, хотите ли вы сохранить конфигурацию ядра — то, что нам и нужно. Просто нажмите кнопку **Yes** (рис. 13.3), затем вы увидите сообщение, что конфигурация ядра записана (рис. 13.4).

После этого нужно перейти в каталог /root/module и ввести ту самую команду `make`:

```
# make -C /usr/src/linux SUBDIRS=$PWD modules
```

На этот раз все должно пройти без замечаний.

13.6. Тестируем наш модуль

Теперь попробуем вставить модуль в ядро:

```
# insmod ./first.ko
```

Обратите внимание: нужно вставить в ядро именно first.ko, а не first.o. Если вам повезет, то на консоли и в системном журнале вы увидите заветное сообщение. Но что делать, если вам не повезло и вместо приветствия модуля вы увидели сообщение:

```
insmod: error inserting './first.ko": -1 Invalid module format
```

Самое интересное, что весь Гугл пестрит сообщениями об этой ошибке, но ни в одном из руководств по созданию модуля не сказано, что с ней делать! Как всегда, пришлось разбираться с ней самому.

Проблема заключается в том, что модуль ядра должен соответствовать версии ядра. Другими словами, в моем случае ошибка заключалась в том, что я использовал исходные тексты одной версии ядра, а запустить модуль пытался под управлением другой версии ядра (точнее, сравниваются не версии, а сигнатуры версий ядра, но нам от этого не легче). Такая же ошибка могла произойти и у вас. Например, вы изначально установили систему с ядром 2.6.33.3, также были установлены исходные тексты ядра, а ссылка /usr/src/linux указывала на каталог /usr/src/linux-2.6.33.3. Затем вы перешли на новое ядро, скажем, на 2.6.33.5, загрузив сразу уже скомпилированную версию ядра — путем обновления RPM-пакета ядра.

Понятно, что сейчас вы используете версию 2.6.33.5, а исходные тексты вы обновить забыли. Потом без всякой задней мысли вы собираете модуль ядра — компиляция проходит без ошибок, потому что компилятору все равно, какие заголовочные файлы использовать. Но при запуске модуля система обязательно сообщит вам, что ваш модуль имеет неправильный формат — и это не мудрено.

Бывает и другая причина отличия версий ядер. Разработчики дистрибутивов накладывают на оригинальные исходные тексты ядра множество патчей, поэтому могут возникнуть проблемы с компиляцией модулей ядра. Вы, чтобы скомпилировать модуль, загружаете официальные исходные тексты ядра с kernel.org. Все бы хоро-

шо, но ваш модуль является скомпилированным под загруженную версию ядра, но не под ту, которая сейчас используется.

> ***Примечание***
>
> Если не совпадают сигнатуры версий ядра, теоретически модуль можно заставить работать, указав опцию `--force-vermagic` команды `modprobe`, например `modprobe --force-vermagic first`. Но это не выход из положения, и система может работать нестабильно, старайтесь не использовать данную опцию.

Что делать? Если вы уже собрали модуль, то оставьте его как есть. Перейдите в каталог с исходным кодом ядра и соберите ядро и модули заново:

```
# make menuconfig
# make dep
# make modules
# make
# make modules_install install
```

После этого перезагрузите систему, при загрузке из меню загрузчика выберите только что скомпилированное ядро (неплохо бы запомнить номер версии ядра — вы его узнаете из названия каталога /usr/src/linux-x.x.x.x*). После загрузки системы попробуйте вставить ваш модуль. Все должно работать.

Правда, на данном этапе могут возникнуть (а могут и нет — все зависит от дистрибутива) проблемы куда более серьезные. Как уже было отмечено, разработчики дистрибутива накладывают на оригинальные исходные тексты ядра всевозможные патчи. Может так случиться, что после перезагрузки системы с оригинальным ядром (которое вы скачали с kernel.org) система перестанет загружаться или откажут некоторые устройства. Не то что бы я вас пугаю, но вы должны быть морально готовы к такой ситуации.

Вам кажется процесс создания модулей ядра очень сложным? Так оно и есть, но, по сути, в этом нет ничего удивительного — ведь вы создаете часть операционной системы. А создание операционной системы никогда не было легким занятием.

Просмотреть информацию о модуле можно командой `modinfo` (рис. 13.5):

```
# modinfo ./first.ko
```

```
[root@localhost module]# insmod ./first.ko
insmod: error inserting './first.ko': -1 Invalid module format
[root@localhost module]#  modinfo ./first.ko
filename:       ./first.ko
srcversion:     A826FC67F79803C144628DB
depends:
vermagic:       2.6.33.5-desktop-2mnb SMP mod_unload modversions 686
[root@localhost module]# _
```

Рис. 13.5. Ошибка при загрузке модуля и вывод информации о модуле

Посмотрите на рис. 13.5: модуль собран с использованием исходных текстов ядра 2.6.33.5-desktop, а в моей системе установлено ядро 2.6.33.5-server, хотя ошибка с номерами версий произошла по вине самих разработчиков дистрибутива. При

установке пакета kernel-source не указывается сигнатура ядра, но логично было бы предположить, что при установке виртуального RPM-пакета должны быть установлены исходные тексты, соответствующие используемому ядру. Другими словами, раз по умолчанию устанавливается ядро 2.6.33.5-server, то при установке пакета kernel-source должны быть установлены "исходники" именно "серверной" версии ядра. А на самом деле устанавливаются исходные тексты ядра 2.6.33.5-desktop. Вот какой сюрприз подготовили разработчики Mandriva.

Проблема решилась следующим образом. Поскольку перекомпиляция ядра занимает довольно много времени, то проще установить пакет kernel-server-devel-2.6.33.5-2mnb и пересобрать наш модуль, чем компилировать исходные файлы "настольной" версии ядра и оставить модуль в покое.

После установки пакета в /usr/src появился каталог linux-server-2.6.33.5.2mnb, который я для простоты переименовал в /usr/src/server. Для компиляции модуля теперь используется команда:

```
# make -C /usr/src/server SUBDIRS=$PWD modules
```

Посмотрите на рис. 13.6: модуль собрался без ошибок, при установке командой `insmod` не было никаких "ругательств". Поскольку команда `insmod` была введена в терминале, то приветствие модуля мы не увидели, зато оно появилось в /var/log/messages (рис. 13.7).

Информацию о загруженных модулях можно найти в файле /proc/modules. Наш модуль будет самый первый в списке загруженных модулей (рис. 13.8).

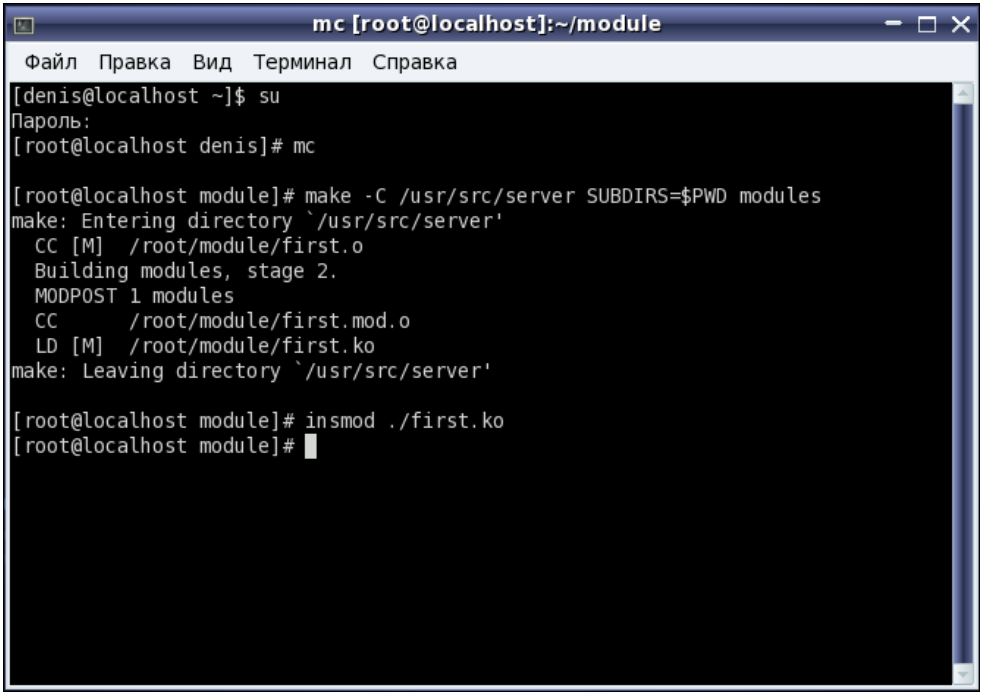


Рис. 13.6. Модуль добавлен в ядро без ошибок

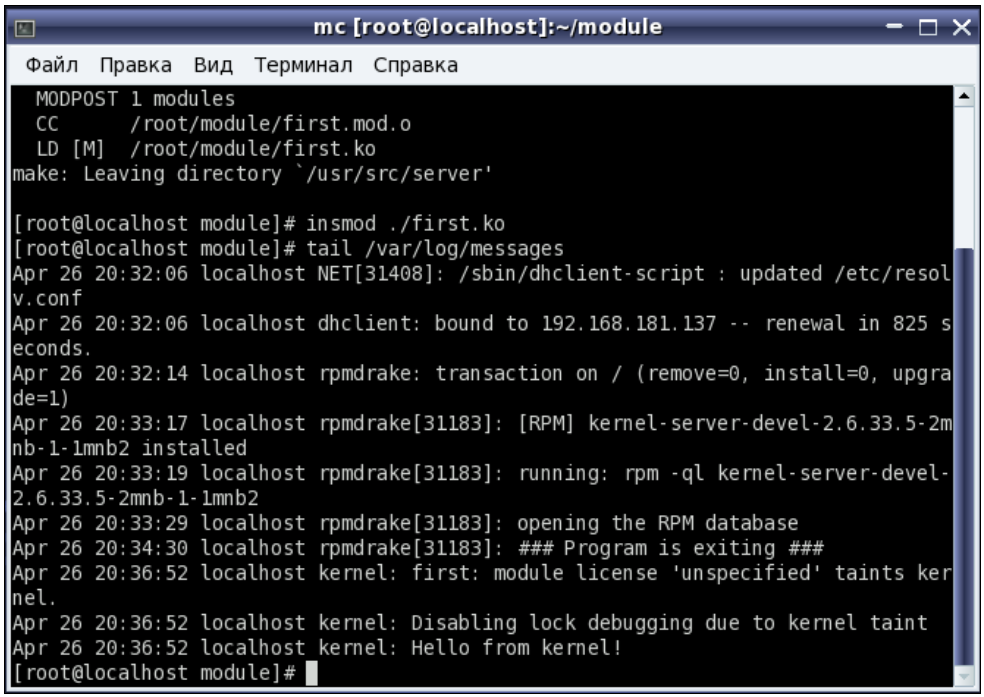


Рис. 13.7. Приветствие модуля

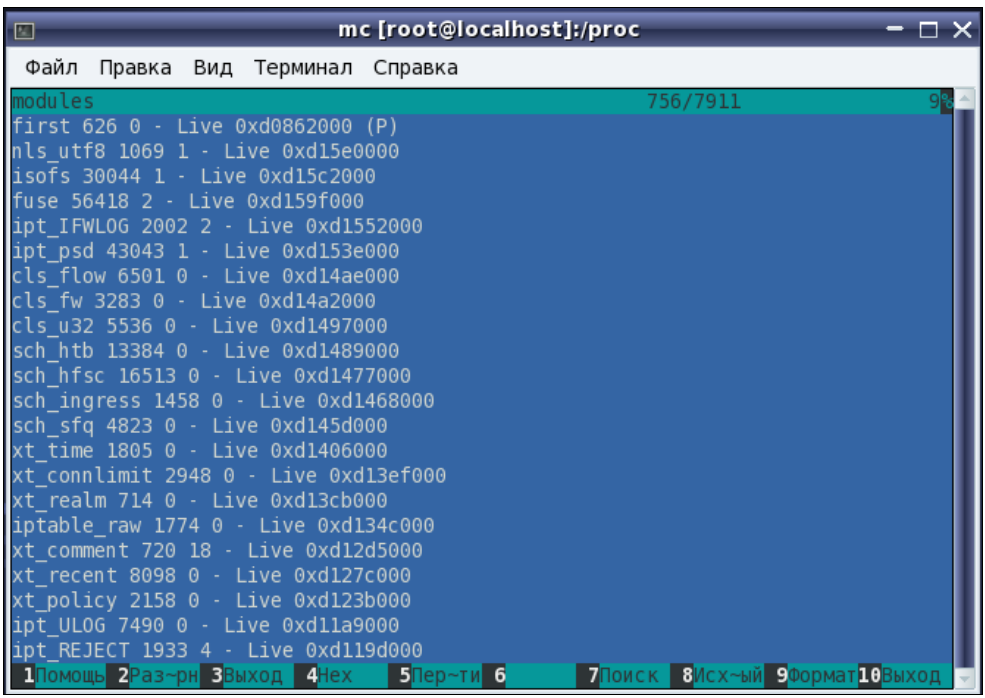


Рис. 13.8. Модуль `first` в списке модулей

Итак, если ваш модуль успешно скомпилировался и запустился, значит, пора принимать поздравления: вы только что написали свой первый модуль ядра!

13.7. Сборка сложных модулей

Сложные модули могут состоять из нескольких исходных файлов. К каждому исходному файлу нужно подключить как минимум два заголовочных файла (а также файлы, содержащие все необходимое для выполняемых модулем операций):

```
#include <linux/kernel.h>
#include <linux/module.h>
```

Представим, что у нас есть два файла с исходным текстом модуля (file1.c и file2.c). Тогда Makefile модуля будет выглядеть так:

```
obj-m += big.o
big-objs := file1.o file2.o
```

13.8. Настоящее программирование ядра

13.8.1. Отличие обычных программ от модулей ядра

Вы думаете, что раз модуль откомпилировался (хотя и с небольшими мучениями), то все самое сложное уже позади? Спешу вас разочаровать: все самое "интересное" только начинается. Ведь до этого мы создали только болванку, не выполняющую никаких действийы — толку от нашего модуля — ноль. Далее мы будем усложнять наш модуль, но сначала вам нужно познакомиться с несколькими страницами теории — без нее двигаться дальше будет проблематично.

Прежде всего, нужно понимать разницу между обычными пользовательскими программами и модулями ядра. В пользовательских программах для вывода сообщений на экран мы привыкли использовать функцию `printf()`. Данная функция определена в стандартной библиотеке libc.

Модуль не может использовать функции обычных библиотек. Он может использовать только функции, экспортируемые ядром. Тут все просто: модуль — часть ядра, и он должен обращаться к ядру, а не к библиотекам пользовательского уровня. Для выполнения обычных операций модуль должен использовать только системные вызовы ядра.

Чтобы просмотреть список функций, экспортируемых ядром, загляните в файл /proc/kallsyms.

В чем заключается разница между системным вызовом и библиотечной функцией? Данный пример уже был приведен в этой книге, но мы рассмотрим его еще раз. Скомпилируйте самую простую программу вроде "Hello, world". А потом просмотрите, сколько системных вызовов использует эта простая программа, выводящая всего одну строку:

```
strace hello-world
```

Впечатляет? А за вывод на экран отвечает системный вызов `write()` — обратите на него внимание в выводе `strace`. Стандартная библиотека позволяет обычным программистам не вникать во все системные подробности и просто писать программы. Теперь вы понимаете разницу между библиотечной функцией и системным вызовом?

Теперь идем дальше. Модуль может переопределять системные вызовы, например вы можете переопределить системный вызов `write()` и переделать его так, что при каждом выводе на стандартный вывод будет добавляться строка `"write()"`. От такой переделки толку мало, но это лишь пример. Можно переопределить любой системный вызов. Цели могут быть разные, например скрыть вредоносную программу из списка процессов — чтобы никто не смог увидеть ее с помощью команд `ps`, `top` и др.

13.8.2. Пространства, пространства и еще раз пространства

За доступ к ресурсам системы (читайте — устройствам) отвечает ядро. Программы конкурируют между собой за доступ к ресурсам системы. Какая именно программа получит в конкретный момент времени доступ к устройству — решать только ядру. Ядро выстраивает конкурирующие запросы в порядке очередности и обрабатывает их. Чтобы получить такую власть над системой и решать, кому и когда будет "место под солнцем", ядро должно работать в *привилегированном режиме* процессора (его же называют нулевым кольцом). Пользовательские программы выполняются в *пользовательском режиме*. Программа обращается к библиотечной функции в пользовательском режиме. Затем библиотечные функции обращаются к системным вызовам. Получается, что хотя системные вызовы и используются обычными пользовательскими функциями, но выполнение операций все равно происходит в привилегированном режиме, т. к. системные вызовы являются частью ядра. Когда системный вызов завершает работу, он возвращает результат в библиотечную функцию, а та, в свою очередь, — в пользовательскую программу.

Получается, что у нас есть два пространства — пространство ядра и пространство пользователя. Эти два пространства не только относятся к режимам выполнения, но и к оперативной памяти — существует два разных адресных пространства.

Операционная система производит переключения из одного пространства пользователя в пространство ядра всякий раз, когда приложение явно или неявно вызывает системный вызов. Код ядра, отвечающий за системный вызов, работает в контексте процесса — от имени вызывавшего процесса — и имеет доступ к данным в адресном пространстве процесса.

Получается вот что. Хотя код ядра и находится в пространстве ядра, но при вызове его программой он работает только с адресным пространством вызвавшего его процесса. Это позволяет обезопасить систему. Ведь в противном случае обычная программа могла вызвать системный вызов, который мог разнести всю систему, т. к. выполняется в привилегированном режиме. Однако этого не произойдет —

все, что может разнести такой системный вызов — это только данные процесса. Конечно, все это верно, если речь идет не о процессе, запущенным с полномочиями root.

Если программа выполняется в пространстве пользователя и только может немного коснуться святая святых — системных вызовов, то модуль изначально выполняется в пространстве ядра, более того, модуль является частью ядра.

Что же касается самого управления памятью, то это очень сложная тема. Могу сказать это еще раз: очень сложная тема. Мне очень жаль, что книг по управлению памятью в Linux практически нет, а рассмотрение модели управления памятью выходит за рамки этой книги. Может, когда-нибудь напишу книгу о том, как Linux управляет памятью. А пока могу порекомендовать вам книгу "Understanding the Linux Kernel" издательства O'Reilly Media. Все бы хорошо, но эта книга довольно стара: ей уже более 10 лет и проблема не в том, что ее будет сложно найти (в электронном виде, думаю, эта книга где-нибудь есть), а в том, что в 2000 году было актуальным ядро версии 2.2, а сейчас — 2.6.

13.8.3. Драйверы устройств и ядро

Есть две причины, по которым вы начали читать эту главу: любопытство и желание разработать собственный драйвер. Обе причины достойны, но сейчас мы поговорим именно о драйверах устройств.

Представим, что у вас есть помощник и вы его попросили доставить какие-то документы вашему партнеру. Вас не интересует, как он это сделает — будет ли он добираться на метро, на машине или просто пешком. Вам важно, чтобы он выполнил ваше задание — доставил документы.

Так вот, драйвер устройства — это ваш помощник. Ядро не знает особенностей устройства, но оно может подать стандартные команды, которые должны принимать все драйверы подобного рода устройств. Например, ядро подает команду уменьшить громкость — драйвер звуковой платы примет команду и выполнит ее (как он это сделает — мы не знаем, все зависит от самого устройства). Понятно, что нельзя отправить команду уменьшения громкости жесткому диску.

При разработке драйверов устройства вам нужно знать, что такое младший и старший номер устройства. Выполните команду:

```
ls -l /dev/sda[1-2]
```

Если на вашем жестком диске больше, чем два раздела (например, четыре раздела), в скобках можете указать другие числа (`1-4`). Результат выполнения этой команды изображен на рис. 13.9. Взгляните на числа, разделенные запятой: старший номер (8) одинаковый для каждого устройства, поскольку оба устройства (/dev/sda1 и /dev/sda2) обслуживаются одним и тем же драйвером. После запятой следует младший номер устройства.

Итак, старший номер — это номер драйвера. У каждого драйвера свой уникальный номер. Младший номер используется драйвером, чтобы различать устройства, ко-

торыми он управляет. Поскольку у каждого устройства разный младший номер, то драйвер "видит" их как разные аппаратные устройства, а не как одно целое.

Как мы уже знаем, все устройства можно разделить на две группы: блочные и символьные. С блочными устройствами обмен данными осуществляется блоками (наборами байтов), а с символьными — посимвольно. Посмотрите на рис. 13.9: перед правами доступа (перед буквой r) вы видите букву b (от англ. *block*). Это означает, что устройство блочное, если же на этом месте будет буква c (от англ. *char*), то перед вами символьное устройство.

```
[root@localhost proc]# ls -l /dev/sda[1-2]
brw-rw---- 1 root disk 8, 1 2011-04-26 13:43 /dev/sda1
brw-rw---- 1 root disk 8, 2 2011-04-26 13:42 /dev/sda2
[root@localhost proc]#
```

Рис. 13.9. Старший и младший номера устройств

Создать устройство можно командой `mknod`, например:

```
mknod /dev/sda3 b 8 3
```

Вы только что создали еще одно блочное устройство (b) /dev/sda3, старший номер устройства равен 8, младший — 3. Файлы устройств не обязательно размещать в каталоге /dev — вы можете выбрать любой другой, но так требует традиция.

Понятие устройства в Linux очень абстрактное. Хотя мы и создали файл устройства /dev/sda3, это не означает, что на нашем жестком диске появился еще один раздел: ведь этот файл никак не связан с аппаратной частью. Да и никак не будет связан, поскольку драйвер жесткого диска "увидит", что третьего раздела на жестком диске нет, и попросту не будет использовать "лишний" файл.

13.9. Символьные устройства

13.9.1. Возможные операции

Драйвер может выполнять какие-то операции с устройством. Операции стандартизированы и описаны в структуре `file_operations` в файле linux/fs.h. Данная структура содержит указатели функций драйвера, которые используются для выполнения разных операций с устройством. Понятно, что драйвер не должен реализовать все функции. Функции, не реализованные драйвером, заменяются пустым указателем `NULL`. Рассмотрим саму структуру:

```
struct file_operations {
struct module *owner;
loff_t(*llseek) (struct file *, loff_t, int);
ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
ssize_t(*write) (struct file *,
        const char __user *, size_t, loff_t *);
```

```
ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t,
                     loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int,
              unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                 loff_t *);
ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                  loff_t *);
ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                    void __user *);
ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                    loff_t *, int);
unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                unsigned long, unsigned long,
                unsigned long);
};
```

Обратите внимание на название функций: они аналогичны знакомым нам системным вызовам. Представим, что вы желаете использовать не все функции, а только `read`, `write`, `open` и `release`, тогда структура `file_operations` может быть заполнена так:

```
struct file_operations fops = {
        .read = mydevice_read,
        .write = mydevice_write,
        .open = mydevice_open,
        .release = mydevice_release
};
```

Обратите внимание на название структуры — `fops`. Указатель на структуру типа `file_operations` обычно называется именно `fops` — так программисты договорились между собой. Конечно, вы можете использовать и другое название — ничего страшного не случится. Но когда в Интернете встретите фрагмент кода с указателем `fops`, знайте, это указатель на структуру `file_operations`.

Каждое устройство в системе представлено в виде файла. Именно поэтому структура операций над устройством называется `file_operations`. А само устройство (т. е. его файл) описывается структурой `file`, которая тоже представлена в linux/fs.h. Указатель на структуру `file` обычно называется `filp`.

13.9.2. Регистрация устройства

Помните, я говорил, что мало создать файл устройства? Нужно его еще связать с самим устройством, т. е. зарегистрировать в ядре. Регистрация говорит ядру, что данный файл устройства будет обслуживаться таким-то драйвером.

При добавлении драйвера происходит его регистрация в ядре и получение старшего номера драйвера. Для регистрации драйвера символьного устройства используется функция `register_chrdev()`, определенная в linux/fs.h:

```
int register_chrdev(unsigned int major,
   const char *name, struct file_operations *fops);
```

Первый параметр — запрашиваемый старший номер устройства, второй — имя файла устройства, третий — указатель на структуру типа `file_operations`. Функции не передается младший номер устройства — ведь это не забота ядра, об обслуживании устройств и назначении им младших номеров должен заботиться сам драйвер.

Как узнать, какой старший номер не используется? Существуют два способа. Первый заключается в использовании динамического номера — просто задайте 0 в качестве первого параметра и ядро само назначит вашему драйверу первый неиспользуемый номер. Второй способ менее правильный — загляните в Documentation/devices.txt, посмотрите, какие номера устройств заняты, и установите незанятый номер. У этого способа есть огромный недостаток: если вы выберете номер, скажем, 77, а потом этот номер будет официально закреплен за другим устройством, ваш драйвер работать не будет.

В случае с динамическим номером тоже есть один неприятный момент: вы не знаете наперед, какой номер устройства будет вам выделен, следовательно, не можете создать файл устройства (ведь при создании нужно указать старший номер устройства). Лучшее решение этой проблемы — использовать системный вызов `mknod()` для создания файла устройства — сразу после получения старшего номера устройства от ядра. Только не забудьте в функции `cleanup_module()` удалить файл устройства.

13.9.3. Драйвер абстрактного символьного устройства

Сейчас мы напишем драйвер абстрактного (несуществующего в природе) символьного устройства. При запуске модуля мы получим старший номер устройства и запишем его в системный журнал /var/log/messages. Дабы не усложнять код модуля, мы не будем вызывать системный вызов `mknod()`: после запуска модуля вы вручную создадите устройство, указав полученный старший номер как аргумент команды `mknod`.

Наш модуль будет поддерживать операции открытия, чтения и освобождения устройства. Операция записи устройства будет поддерживаться частично: вместо полноценной функции будет заглушка.

Код модуля представлен в листинге 13.4.

Листинг 13.4. Модуль mydev.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* Функция put_user */

/* Для большего удобства определяем прототипы функций */
int init_module(void);
void cleanup_module(void);
static int mydevice_open(struct inode *, struct file *);
static int mydevice_release(struct inode *, struct file *);
static ssize_t mydevice_read(struct file *, char *, size_t, loff_t *);
static ssize_t mydevice_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "mydev"         /* Имя устройства */
#define BUF_LEN 128                 /* Максимальная длина сообщения */

static int Major;                   /* Старший номер устройства */
static int Device_Status = 0;       /* Статус устройства, 1 = открыто */
static char msg[BUF_LEN];           /* Буфер для сообщения */
static char *msg_Ptr;

/* Структура операций */
static struct file_operations fops = {
  .open = mydevice_open,
  .release = mydevice_release,
  .read = mydevice_read,
  .write = mydevice_write
};


/* Инициализация модуля */
int init_module(void)
{
  /* Пытаемся зарегистрировать устройство */
  Major = register_chrdev(0, DEVICE_NAME, &fops);

  if (Major < 0) {
    printk(KERN_ALERT "register_chrdev() error %d\n", Major);
    return Major;
  }

  printk(KERN_NOTICE "Major number %d\n", Major);
  printk(KERN_NOTICE "Please create a dev file with\n");
  printk(KERN_NOTICE "'mknod /dev/mydev c %d 0'.\n", Major);
```

```
  return SUCCESS;
}

void cleanup_module(void)
{
  /* Удаляем устройство */
  int res = unregister_chrdev(Major, DEVICE_NAME);
  if (res < 0)
    printk(KERN_ALERT "unregister_chrdev() error: %d\n", res);
}

/* Обработчики устройства */

/* Обработчик открытия устройства */
static int mydevice_open(struct inode *inode, struct file *file)
{
  if (Device_Status)
    return -EBUSY;
  Device_Open++;
  sprintf(msg, "Hello\n");
  msg_Ptr = msg;
  try_module_get(THIS_MODULE);

  return SUCCESS;
}

/* Обработчик закрытия устройства */
static int mydevice_release(struct inode *inode, struct file *file)
{
  Device_Status--;
  module_put(THIS_MODULE);
  return SUCCESS;
}

/* Обработчик записи устройства */
static ssize_t
mydevice_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
  printk(KERN_ALERT "Unsupported operation: write()\n");
  return -EINVAL;
}

/* Обработчик чтения устройства — вызывается, когда процесс пытается прочитать
уже открытый файл.
filp — указатель на структуру file
buffer — буфер, в который нужно занести данные
lenght — длина буфера
offset — смещение */
```

```
mystatic ssize_t mydevice_read(struct file *filp, char *buffer, size_t length,
loff_t * offset)
{

  int bytes_read = 0; /* Сколько байтов записано в буфер */


  if (*msg_Ptr == 0)          /* Достигнут конец сообщения,
                              возвращаем 0 как признак конца файла */
    return 0;

  /* Самый важный момент: перемещение данных. Поскольку буфер
находится в сегменте данных пользовательского процесса
(в пространстве пользователя), а не в пространстве ядра,
то мы не можем выполнить простое присваивание. Чтобы
скопировать данные, нам нужно использовать функцию put_user(),
которая перенесет данные из пространства ядра в пространство пользователя
 */
  while (length && *msg_Ptr) {

    put_user(*(msg_Ptr++), buffer++);

    length--;
    bytes_read++;
   }

  /* Возвращаем количество реально записанных в буфер байтов */
  return bytes_read;
}

MODULE_LICENSE("GPL");                              /* Лицензия */
MODULE_AUTHOR("Denis Kolisnichenko");               /* Автор */
MODULE_DESCRIPTION("Driver for /dev/mydev");        /* Описание */
MODULE_SUPPORTED_DEVICE("mydev");           /* Устройство */
```

Ничего не понятно? Давайте разберем наш модуль по кирпичику. Мы подключаем заголовочные файлы:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* Функция put_user */
```

Первые два стандартны для любого модуля. В третьем находится структура `file_operations()`, необходимая для переопределения обработчиков устройства. В последнем находится функция `put_user()`, передающая данные из пространства ядра в пространство пользователя.

Далее объявляем три константы и четыре статических переменных:

```
#define SUCCESS 0
#define DEVICE_NAME "mydev"         /* Имя устройства */
#define BUF_LEN 128                 /* Максимальная длина сообщения */

static int Major;                   /* Старший номер устройства */
static int Device_Status = 0;       /* Статус устройства, 1 = открыто */
static char msg[BUF_LEN];           /* Буфер для сообщения */
static char *msg_Ptr;
```

Первая константа нужна только для красоты — вместо нее можно было бы просто передавать 0, если вызов обработчика устройства завершился успешно. Вторая константа задает имя устройства, а третья — длину буфера сообщений.

В переменной `Major` будет храниться старший номер устройства. Переменная `Device_Status` хранит статус устройства: 1 — открыто (уже используется каким-то процессом), 0 — свободно. Переменная `msg` содержит буфер для сообщения, которое получит процесс, когда попытается прочитать устройство /dev/mydev. Последняя переменная — указатель на буфер.

При инициализации устройства мы получаем его старший номер — заполняем переменную `Major`. Если зарегистрировать устройство не удалось, модуль прекращает свою работу — функция инициализации возвращает значение, отличное от 0, а именно — код возврата функции `register_chrdev()`. Если же все прошло успешно, модуль сообщает старший номер устройства, чтобы вы могли создать само устройство командой `mknod`. Обратите внимание: критические ошибки (когда модуль больше не может продолжить работу) выводятся как `KER_ALERT` (будут видны на консоли), а все остальные сообщения — как `KERN_NOTICE` (будут видны только в системном журнале):

```
  Major = register_chrdev(0, DEVICE_NAME, &fops);

  if (Major < 0) {
    printk(KERN_ALERT "register_chrdev() error %d\n", Major);
    return Major;
  }

  printk(KERN_NOTICE "Major number %d\n", Major);
  printk(KERN_NOTICE "Please create a dev file with\n");
  printk(KERN_NOTICE "'mknod /dev/mydev c %d 0'.\n", Major);
```

При завершении работы модуля мы должны удалить регистрацию устройства (файл, созданный командой `mknod`, останется в каталоге /dev, но связь между файлом устройства и драйвером будет аннулирована):

```
int res = unregister_chrdev(Major, DEVICE_NAME);
```

После определения двух главных функций модуля можно приступить к написанию обработчиков устройства, которые объявлены в структуре `filp`:

```
static struct file_operations fops = {
  .open = mydevice_open,
  .release = mydevice_release,
  .read = mydevice_read,
  .write = mydevice_write
};
```

Прежде всего, разберемся, когда вызывается тот или иной обработчик:

- `mydevice_open()` — вызывается, когда процесс пытается открыть файл;
- `mydevice_release()` — вызывается при закрытии файла;
- `mydevice_read()` — вызывается, когда процесс пытается прочитать файл устройства;
- `write()` — когда процесс пытается записать что-то в файл устройства, например `echo info > /dev/mydev`.

При открытии устройства мы должны проверить его статус: если он равен 1, значит, устройство уже используется другим процессом, и мы возвращаем значение `-EBUSY`: устройство занято. Если же статус устройства равен 0, тогда мы увеличиваем статус (все — с этого момента устройство занято) и заполняем буфер текстовой строкой.

```
if (Device_Status)
    return -EBUSY;
  Device_Open++;
  sprintf(msg, "Hello\n");
  msg_Ptr = msg;
```

При закрытии устройства мы должны установить статус устройства в 0:

```
Device_Status--;
```

Функция записи вообще элементарна — это просто заглушка. Мы выводим сообщение о том, что операция записи не поддерживается. На этом вся запись заканчивается. А вот функция чтения устройства заслуживает внимания. Обычно все функции чтения возвращают количество прочитанных байтов, что будет делать и наша функция, поэтому она начинается с переменной `bytes_read`. Далее в цикле `while` мы посимвольно передаем содержимое нашего буфера пользовательскому процессу с помощью функции `put_user()`. Также в цикле `while` увеличивается переменная `bytes_read`, которая потом и возвращается пользовательскому процессу:

```
int bytes_read = 0;          /* Сколько байтов записано в буфер */

  if (*msg_Ptr == 0)         /* Достигнут конец сообщения, возвращаем 0
                             как признак конца файла */
    return 0;

   while (length && *msg_Ptr) {

    put_user(*(msg_Ptr++), buffer++);
```

```
    length--;
    bytes_read++;
   }

  return bytes_read;
```

Вот теперь, я думаю, все стало на свои места. Попробуем усложнить наш модуль, добавив поддержку записи. Для этого нужно изменить всего лишь одну функцию — обработчик записи устройства. Если функция чтения устройства передавала пользовательскому процессу информацию с помощью функции `put_user()`, то сейчас мы будем получать информацию от пользовательского процесса с помощью функции `get_user()`:

```
static ssize_t
mydevice_write(struct file *file,
            const char __user * buffer, size_t length, loff_t * offset)
{
  int bwrite;

  for (bwrite = 0; bwrite < length && bwrite < BUF_LEN; bwrite ++)
    get_user(Message[i], buffer + i);

  Message_Ptr = Message;

  /* Возвращаем к-во принятых байтов */
  return bwrite;
}
```

Все бы хорошо, и наше абстрактное виртуальное устройство отлично работает. Но на практике вам придется столкнуться с управлением самим физическим устройством. В зависимости от специфики устройства, возможно, вам придется использовать функцию `ioctl()`, ее название является сокращением от Input/Output ConTroL. Поскольку я не знаю специфики вашего устройства, приводить какой-либо код — глупо. Возможно, он подойдет одному читателю из 1000, а может, наоборот, будет интересен 1000 читателям, но не лично вам. При использовании `ioctl` структура операций будет выглядеть так:

```
struct file_operations fops = {
  .read = mydevice_read,
  .write = mydevice_write,
  .ioctl = mydevice_ioctl,
  .open = mydevice_open,
  .release = mydevice_release,
};
```

13.10. Создание файла в /proc

Файловая система /proc используется для предоставления информации о процессах и о системе, например в файле /proc/modules содержится список загруженных мо-

дулей. Некоторые файлы в /proc поддерживают не только запись, но и чтение и могут даже использоваться для настройки системы на лету, но сейчас нас такие файлы не интересуют. Попробуем создать файл /proc/mydev, предоставляющий информацию о нашем устройстве.

Для работы с /proc нужно подключить заголовочный файл proc_fs.h. Но поскольку мы все еще разрабатываем модуль ядра, то нам нужны файлы module.h и kernel.h:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
```

Для работы с /proc нам нужна структура `proc_dir_entry`:

```
struct proc_dir_entry *pde;
```

Структура `proc_dir_entry` заполняется так:

```
  /* имя файла и права доступа */
  pde = create_proc_entry("mydev", 0644, NULL);
  pde->read_proc = proc_read;              /* Обработчик чтения */
  pde->owner = THIS_MODULE;          /* Владелец proc-файла */
  pde->mode = S_IFREG | S_IRUGO;     /* Режим */
  pde->uid = 0;                                    /* UID */
  pde->gid = 0;                                    /* GID */
  pde->size = 50;                            /* Размер */
```

Теперь рассмотрим функцию `proc_read()` — обработчик чтения proc-файла:

```
ssize_t
proc_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
```

Ей нужно передать шесть параметров:

- буфер с данными — сюда можно записать все что угодно;
- указатель на строку с данными — если вы не хотите использовать параметр `buffer`;
- смещение — текущая позиция в файле;
- длина буфера — собственно, длина как она есть;
- признак конца файла — если `eof = 1`, достигнут конец файла;
- указатель на данные — нужен, только если у вас один обработчик, а proc-файлов — несколько.

Код функции `proc_read` может быть таким:

```
ssize_t
proc_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
```

```
{
  int length = 0;             /* Число байт */
  static int c = 1;          /* Сколько раз был прочитан файл */

  /* Достигнут конец файла, нужно сообщить об этом процессу,
  читающему файл */
  if (offset > 0) {
    printk(KERN_INFO "Offset %d, Bytes %d\n", (int)(offset), length);
    *eof = 1;
    return length;
  }

  /* Заполняем буфер */
  length = sprintf(buffer,
          "This file has been read %d times\n", c);
  c++;

  return length;
}
```

Функция инициализации модуля заполняет структуру `pde`:

```
int init_module()
{
  int res = 0;
  pde = create_proc_entry("mydev", 0644, NULL);
  pde->read_proc = proc_read;
  pde->owner = THIS_MODULE;
  pde->mode = S_IFREG | S_IRUGO;
  pde->uid = 0;
  pde->gid = 0;
  pde->size = 50;

  printk(KERN_INFO "Module started. Creating /proc/mydev...");

  if (pde == NULL) {
    res = -1;
    remove_proc_entry("mydev", &proc_root);
    printk(KERN_INFO "Error\n");
  } else
    printk(KERN_INFO "OK\n");

  return res;
}
```

Если вызов `create_proc_entry` не удался, то мы удаляем наш proc-файл вызовом `remove_proc_entry()` и завершаем работу модуля.

Функция `cleanup_module()` очень проста и выглядит так:

```
void cleanup_module()
{
  remove_proc_entry("mydev", &proc_root);
  printk(KERN_INFO "Module stopped\n");
}
```

Теперь нам осталось собрать все вместе (листинг 13.5).

Листинг 13.5. Модуль procf.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>

struct proc_dir_entry *pde;

ssize_t
proc_read(char *buffer, char **buffer_location,
          off_t offset, int buffer_length, int *eof, void *data)
{

  int length = 0;             /* Число байт */
  static int c = 1;           /* Сколько раз был прочитан файл */

  /* Достигнут конец файла, нужно сообщить об этом процессу,
  читающему файл */
  if (offset > 0) {
    printk(KERN_INFO "Offset %d, Bytes %d\n", (int)(offset), length);
    *eof = 1;
    return length;
  }

  /* Заполняем буфер */
  length = sprintf(buffer,
          "This file has been read %d times\n", c);
  c++;

  return length;
}
int init_module()
{
  int res = 0;
  pde = create_proc_entry("mydev", 0644, NULL);
  pde->read_proc = proc_read;
  pde->owner = THIS_MODULE;
  pde->mode = S_IFREG | S_IRUGO;
```

```
  pde->uid = 0;
  pde->gid = 0;
  pde->size = 50;

  printk(KERN_INFO "Module started. Creating /proc/mydev...");

  if (pde == NULL) {
    res = -1;
    remove_proc_entry("mydev", &proc_root);
    printk(KERN_INFO "Error\n");
  } else
    printk(KERN_INFO "OK\n");

  return res;
}

void cleanup_module()
{
  remove_proc_entry("mydev", &proc_root);
  printk(KERN_INFO "Module stopped\n");
}
```

Скомпилируйте и установите модель (командой `insmod`). После этого загляните в каталог /proc — в нем будет файл mydev. Откройте его и просмотрите содержимое.

13.11. Полезный пример: клавиатурный шпион

Вы прочитали уже много страниц. Как уже отмечалось, разработка модулей заслуживает отдельной книги, но вам, как читателю, наверное, хочется какого-то полезного примера. А то до сих пор кроме пары бестолковых модулей вы ничего не научились создавать.

Наверняка все пользовались особыми программами — клавиатурными снифферами. Такие программы также называют клавиатурными шпионами (key loggers). Клавиатурный шпион запускается незаметно — его никто не видит, но он есть. Он перехватывает все нажатия клавиш и записывает их в отдельный файл, анализируя который, можно понять, какие клавиши нажимал пользователь.

Мы реализуем наш клавиатурный шпион в виде модуля ядра — идеальная среда для написания подобного рода программ. Наш шпион будет записывать информацию о нажатых клавишах не в отдельный файл, а в системный журнал — для упрощения кода модуля.

На примере клавиатурного шпиона будет продемонстрирована обработка прерываний (IRQ) и очереди задач. Наш модуль будет содержать обработчик прерывания 1 (это и есть прерывание клавиатуры на архитектуре Intel). Вы только подумайте:

средняя скорость печати опытного пользователя на клавиатуре составляет 100–250 символов в минуту. Следовательно, каждую минуту в системный журнал будет записываться до 250 записей, что может отрицательно сказаться на производительности системы. Поэтому и используем очереди: каждую минуту будет до 250 раз вызываться обработчик прерывания 1, а запись, благодаря очереди задач, в системный журнал будет происходить тогда, когда удобно ядру.

Для реализации нашего шпиона нужно подключить следующие заголовочные файлы:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <asm/io.h>
```

Затем нужно определить очередь заданий `myqueue`:

```
#define MY_WORK_QUEUE_NAME "WQsched.c"
static struct workqueue_struct *myqueue;
```

Некоторые функции, относящиеся к очереди заданий, работают, только если модуль лицензирован под GPL, поэтому нам ничего не остается, как объявить это:

```
MODULE_LICENSE("GPL");
```

В процессе инициализации модуля мы заменим оригинальный обработчик прерывания клавиатуры (IRQ 1). Логично было бы предположить, что при удалении модуля мы должны восстановить оригинальный обработчик прерывания, но поскольку это невозможно (оригинальный обработчик будет восстановлен только после клавиатуры), то функция `cleanup_module()` не будет ничего делать:

```
void cleanup_module()
{
}
```

Функция инициализации модуля выглядит так:

```
int init_module()
{
  /* Создаем очередь */
  myqueue = create_workqueue(MY_WORK_QUEUE_NAME);

  /* Освобождаем старый обработчик прерывания IRQ 1 */
        free_irq(1, NULL);

  return request_irq(1,              /* Номер IRQ */
         keyboard_handler,           /* наш обработчик */
         SA_SHIRQ,
         "keyboard_irq_handler",
         (void *)(keyboard_handler));
}
```

Первым делом функция `init_module()` создает очередь, затем она освобождает стандартный обработчик прерывания 1 с помощью вызова `free_irq()`. Потом мы назначаем новый обработчик прерывания с помощью вызова `request_irq()`. Первый параметр этого вызова — номер прерывания, затем идет название функции обработчика. Флаг `SA_SHIRQ` означает, что это прерывание может совместно обслуживаться другими обработчиками. Следующий параметр — название нашего обработчика, которое будет отображено в /proc/interrupts. Последний параметр — ID устройства. Если вы не используете флаг `SA_SHIRQ`, установите его в `NULL`, в противном случае нужно указать адрес функции-обработчика прерывания.

Теперь рассмотрим функцию `keyboard_handler()`. Задача этой функции — получить состояние клавиатуры и скан-код нажатой клавиши и передать все это функции `log_char()`, которая обеспечит протоколирования нажатой клавиши. Вот только вместо прямого вызова `log_char()` функция `keyboard_handler()` добавляет вызов `log_char()` в очередь заданий. Для формирования задания используется структура `work_struct`. В качестве задания выступает вызов `log_char()` с передачей ему параметра — скан-кода клавиши. Статус клавиатуры передавать не будем, дабы не захламлять журнал лишней информацией.

```
irqreturn_t keyboard_handler(int irq, void *dev_id, struct pt_regs *regs)
{

  static int initialised = 0;
  static unsigned char scancode;
  static struct work_struct mytask;
  unsigned char status;

  /*  Читаем состояние клавиатуры и скан-код клавиши */
  status = inb(0x64);
  scancode = inb(0x60);

  /* Формируем задачу mytask — ею будет функция log_char */
  if (initialised == 0) {
    INIT_WORK(&mytask, log_char, &scancode);
    initialised = 1;
  } else {
    PREPARE_WORK(&mytask, log_char, &scancode);
  }

  /* Добавляем задачу mytask в очередь myqueue */
  queue_work(myqueue, &mytask);

  return IRQ_HANDLED;
}
```

Функция `log_char()` очень проста. Ее задача — записать в системный журнал скан-код клавиши. Для облегчения поиска строк, выводимых сниффером в журнал, я добавил маркер `=007=`. После этого вывести все строки можно будет так:

```
cat /var/log/messages | grep =007= | less
```

Вот код самой функции:

```
static void log_char(void *scancode)
{
  printk(KERN_INFO "=007=: Scancode %x %s.\n",
         (int)*((char *)scancode) & 0x7F,
         *((char *)scancode) & 0x80 ? "Released" : "Pressed");
}
```

В листинге 13.6 представлен полный код модуля без лишних комментариев.

Листинг 13.6. Модуль sniffer.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <asm/io.h>

#define MY_WORK_QUEUE_NAME "WQsched.c"

static struct workqueue_struct *myqueue;

MODULE_LICENSE("GPL");

static void log_char(void *scancode)
{
  printk(KERN_INFO "=007=: Scancode %x %s.\n",
         (int)*((char *)scancode) & 0x7F,
         *((char *)scancode) & 0x80 ? "Released" : "Pressed");
}

irqreturn_t keyboard_handler(int irq, void *dev_id, struct pt_regs *regs)
{

  static int initialised = 0;
  static unsigned char scancode;
  static struct work_struct mytask;
  unsigned char status;

  /*  Читаем состояние клавиатуры и скан-код клавиши */
  status = inb(0x64);
  scancode = inb(0x60);

  /* Формируем задачу mytask — ею будет функция log_char */
  if (initialised == 0) {
    INIT_WORK(&mytask, log_char, &scancode);
```

```
    initialised = 1;
  } else {
    PREPARE_WORK(&mytask, log_char, &scancode);
  }

  /* Добавляем задачу mytask в очередь myqueue */
  queue_work(myqueue, &mytask);

  return IRQ_HANDLED;
}


int init_module()
{
  /* Создаем очередь */
  myqueue = create_workqueue(MY_WORK_QUEUE_NAME);

  /* Освобождаем старый обработчик прерывания IRQ 1 */
        free_irq(1, NULL);

  return request_irq(1,              /* Номер IRQ */
         keyboard_handler,           /* Наш обработчик */
         SA_SHIRQ,
         "keyboard_irq_handler",
         (void *)(keyboard_handler));
}

void cleanup_module()
{
}
```

На этом все. Глава получилась довольно большой, но в ней мы не рассмотрели и десятой части всего, с чем можно столкнуться при программировании ядра. Следующая часть книги не будет такой сложной: в ней вы систематизируете свои знания о файловой системе Linux.

ЧАСТЬ IV

Файловая система Linux

Глава 14.	Введение в файловую систему
Глава 15.	Операции над каталогами
Глава 16.	Операции с файлами
Глава 17.	Получение информации о файловой системе
Глава 18.	Права доступа к файлам и каталогам
Глава 19.	Псевдофайловые системы

Полноценно программировать в Linux можно, только если понимаешь устройство и особенности файловой системы Linux. Вся четвертая часть книги посвящена файловой системе Linux. В ней вы не только изучите файловую систему, но и подробнее ознакомитесь с разработкой приложений, использующих файлы для хранения данных.

ГЛАВА 14

Введение в файловую систему

14.1. Родные файловые системы Linux

Linux поддерживает много различных файловых систем. Начинающий пользователь просто теряется, когда видит такое многообразие выбора, — ведь в качестве корневой файловой системы доступны: ext2, ext3, ext4, XFS, ReiserFS, JFS.

Ранее "родной" (native) файловой системой Linux была ext2 (файловая система ext использовалась разве что в самых первых версиях Linux), затем ей на смену пришла *журналируемая* версия файловой системы — ext3. Сегодня все современные дистрибутивы по умолчанию используют следующее поколение файловой системы — ext4. В этой главе помимо всего прочего вы найдете сравнение последних версий файловых систем Linux — ext3 и ext4.

> ***Примечание***
>
> Linux до сих пор поддерживает файловую систему ext, но она считается устаревшей, и рекомендуется воздержаться от ее использования.

Таким образом, в качестве корневой файловой системы и файловой системы других Linux-разделов используются файловые системы ext3, ext4, XFS, ReiserFS, JFS. Все перечисленные файловые системы (кроме ext2) ведут журналы своей работы, что позволяет восстановить данные в случае сбоя. Осуществляется это следующим образом — перед тем как выполнить операцию, журналируемая файловая система записывает эту операцию в *журнал*, а после выполнения операции удаляет запись из журнала. Представим, что после занесения операции в журнал произошел сбой (например, "выключили свет"). Позже, когда сбой будет устранен, файловая система по журналу выполнит все действия, которые в него занесены. Конечно, и это не всегда позволяет уберечься от последствий сбоя — стопроцентной гарантии никто не дает, но все же такая схема работы лучше, чем вообще ничего.

Файловые системы ext2 и ext3 совместимы. По сути, ext3 — та же ext2, только с журналом. Раздел ext3 могут читать программы (например, Total Commander и Ext2Fsd в Windows), рассчитанные на ext2. В свою очередь, ext4 — это усовершенствованная версия ext3.

В современных дистрибутивах по умолчанию задана файловая система ext4 (хотя в некоторых дистрибутивах все еще может использоваться ext3). При необходимости можно выбрать другие файловые системы. Далее мы рассмотрим их особенности, чтобы понять, нужно ли их использовать или же остановить свой выбор на стандартной ext3/ext4.

- *Файловая система XFS* была разработана компанией Silicon Graphics в 2001 году. Основная особенность данной системы — высокая производительность (до 7 Гбайт/с). XFS может работать с блоками размером от 512 байтов до 64 Кбайт. Ясно, что если у вас много маленьких файлов, то в целях экономии места можно установить самый маленький размер блока. А если вы работаете с файлами большого размера (например, мультимедиа), то нужно выбрать самый большой размер блока — так файловая система обеспечит максимальную производительность (конечно, если "железо" позволяет). Учитывая высокую производительность этой файловой системы, ее нет смысла устанавливать на домашнем компьютере, поскольку все ее преимущества будут сведены на нет. А вот если вы будете работать с файлами очень большого размера, XFS проявит себя с лучшей стороны.
- *Файловая система ReiserFS* считается самой экономной, поскольку позволяет хранить несколько файлов в одном блоке (другие файловые системы могут хранить в одном блоке только один файл или одну его часть). Например, если размер блока равен 4 Кбайт, а файл занимает всего 512 байт (а таких файлов очень много в разных каталогах), то 3,5 Кбайт просто не будут использоваться. А вот ReiserFS позволяет задействовать буквально каждый байт жесткого диска!

  Но у этой файловой системы есть два больших недостатка: она неустойчива к сбоям и ее производительность сильно снижается при фрагментации. Поэтому, если вы выбираете данную файловую систему, покупайте UPS (источник бесперебойного питания) и почаще дефрагментируйте жесткий диск.
- *Файловая система JFS* (разработка IBM) сначала появилась в операционной системе AIX, а потом была модифицирована под Linux. Основные достоинства этой файловой системы — надежность и высокая производительность (выше, чем у XFS). Но у нее маленький размер блока (от 512 байтов до 4 Кбайт). Следовательно, она хороша на сервере баз данных, но не при работе с данными мультимедиа, поскольку блок в 4 Кбайт для работы, например, с видео в реальном времени будет маловат.

В этой книге мы не будем подробно рассматривать особенности ext4. Если вы заинтересовались, то рекомендую прочитать одну из моих книг, посвященных настройке Linux, например "Linux. От новичка к профессионалу" (3-е издание) или "Серверное применение Linux (3-е издание)".

14.2. Особенности файловой системы Linux

14.2.1. Имена файлов в Linux

По сравнению с Windows в Linux несколько другие правила построения имен файлов, вам придется с этим смириться. Начнем с того, что в Linux нет такого понятия,

как расширение имени файла. В Windows, например, для файла Document1.doc именем файла является фрагмент Document1, а doc — это расширение. В Linux Document1.doc — это имя файла, никакого расширения нет.

Максимальная длина имени файла — 254 символа. Имя может содержать любые символы (в том числе и кириллицу), кроме / \ ? < > * " |. Но кириллицу в именах файлов я бы не рекомендовал вообще. Впрочем, если вы уверены, что не будете эти файлы передавать Windows-пользователям (на дискете, по электронной почте) — используйте на здоровье. А при обмене файлами по электронной почте имя файла лучше писать латиницей (кодировка-то у всех разная, поэтому вместо русскоязычного имени пользователь увидит абракадабру).

Также вам придется привыкнуть к тому, что Linux чувствительна к регистру в имени файла: FILE.txt и FILE.Txt — это два разных файла.

Разделение элементов пути осуществляется символом / (прямой слэш), а не \ (обратный слэш), как в Windows.

14.2.2. Файлы и устройства

Пользователи Windows привыкли к тому, что файл — это именованная область данных на диске. Отчасти так оно и есть. Отчасти — потому, что приведенное определение файла было верно для DOS (Disk Operating System) и Windows.

В Linux же понятие файла значительно шире. Сейчас Windows-пользователи будут очень удивлены: в Linux есть файлы устройств, позволяющие обращаться с устройством, как с обычным файлом. Файлы устройств находятся в каталоге /dev (от *devices*). Да, через файл устройства мы можем обратиться к устройству! Если вы работали в DOS, то, наверное, помните, что что-то подобное было и там — существовали зарезервированные имена файлов: PRN (принтер), CON (клавиатура при вводе, дисплей при выводе), LPT*n* (параллельный порт, *n* — номер порта), COM*n* (последовательный порт).

> ***Примечание***
>
> Кому-то может показаться, что разработчики Linux "украли" идею специальных файлов у Microsoft — ведь Linux появилась в начале 90-х годов, а DOS — в начале 80-х годов прошлого века. На самом деле это не так. Наоборот, Microsoft позаимствовала идею файлов устройств из операционной системы UNIX, которая была создана еще до создания DOS. Однако сейчас не время говорить об истории развития операционных систем, поэтому лучше вернемся к файлам устройств.

Вот самые распространенные примеры файлов устройств:

- /dev/sd*x* — файл жесткого диска или USB-накопителя;
- /dev/sd*xN* — файл раздела на жестком диске, *N* — это номер раздела;
- /dev/mouse — файл мыши;
- /dev/modem — файл модема (на самом деле является ссылкой на файл устройства ttyS*n*);
- /dev/ttyS*n* — файл последовательного порта, *n* — номер порта (ttyS0 соответствует COM1, ttyS1 — COM2 и т. д.).

Примечание

В современных дистрибутивах имена вида /dev/hd*x* уже не используются *(см. далее)*.

В свою очередь, файлы устройств бывают двух типов: *блочные* и *символьные*. Обмен информацией с блочными устройствами, например с жестким диском, осуществляется блоками информации, а с символьными — отдельными символами. Пример символьного устройства — последовательный порт.

14.2.3. Корневая файловая система и монтирование

Наверняка, на вашем компьютере установлена система Windows. Откройте окно **Компьютер** (или **Мой компьютер** — в Windows XP).

Скорее всего, вы увидите пиктограмму гибкого диска (имя устройства A:), пиктограммы разделов жесткого диска (пусть будет три раздела — C:, D: и E:), пиктограмму привода CD/DVD (F:). Таким способом с помощью буквенных обозначений A:, C:, D: и т. д. в Windows обозначаются корневые каталоги разделов жесткого диска и сменных носителей.

В Linux существует понятие *корневой файловой системы*. Допустим, вы установили Linux в раздел с именем /dev/sda3. В этом разделе и будет развернута корневая файловая система вашей Linux-системы. Корневой каталог обозначается прямым слэшем — /, т. е. для перехода в корневой каталог в терминале (или консоли) нужно ввести команду `cd /`.

Понятно, что на вашем жестком диске есть еще разделы. Чтобы получить доступ к этим разделам, вам нужно *подмонтировать* их к корневой файловой системе. После монтирования вы можете обратиться к содержимому разделов через точку монтирования — назначенный вами при монтировании специальный каталог, например /mnt/cdrom. Монтированию файловых систем посвящен *разд. 14.4.6*, поэтому сейчас не будем говорить об этом процессе подробно.

14.2.4. Стандартные каталоги Linux

Файловая система любого дистрибутива Linux содержит следующие каталоги:

- / — корневой каталог;
- /bin — содержит стандартные программы Linux (cat, cp, ls, login и т. д.);
- /boot — каталог загрузчика, содержит образы ядра и Initrd, может содержать конфигурационные и вспомогательные файлы загрузчика;
- /dev — содержит файлы устройств;
- /etc — содержит конфигурационные файлы системы;
- /home — содержит домашние каталоги пользователей;
- /lib — библиотеки и модули;
- /lost+found — восстановленные после некорректного размонтирования файловой системы файлы и каталоги;

- /misc — может содержать все что угодно, равно как и каталог /opt;
- /mnt — обычно содержит точки монтирования;
- /proc — каталог псевдофайловой системы procfs, предоставляющей информацию о процессах;
- /root — каталог суперпользователя root;
- /sbin — каталог системных утилит, выполнять которые имеет право пользователь root;
- /tmp — каталог для временных файлов;
- /usr — содержит пользовательские программы, документацию, исходные коды программ и ядра;
- /var — постоянно изменяющиеся данные системы, например: очереди системы печати, почтовые ящики, протоколы, замки и т. д.

14.3. Внутреннее строение файловой системы

Что такое файловая система? Можно встретить различные определения, и все они будут правильные. Наиболее точным я считаю следующее:

> Файловая система — это способ представления информации на носителе данных, а также часть операционной системы, обеспечивающая выполнение операций над файлами.

Из приведенного определения ясно, что файловая система состоит из двух частей, двух уровней: уровня представления данных и набора системных вызовов для работы с этими данными.

Любая операционная система может работать с разными файловыми системами, например со своей основной файловой системой и с файловой системой компакт-дисков (ISO 9660). Задача операционной системы заключается в предоставлении стандартного интерфейса, позволяющего обращаться к каждой файловой системе, не обращая внимания на ее особенности. Например, в Linux для открытия файла используется системный вызов `open()` — программа просто вызывает `open()`, передав ему имя файла, а на какой файловой системе расположен этот файл — дело третье.

Рассмотрим схему архитектуры файловой системы (рис. 14.1): верхние два элемента — это пользовательский уровень, все последующие — уровень ядра.

Приложение может использовать функции glibc (библиотека GNU C) или же напрямую системные вызовы ядра — тут уж как будет угодно программисту. Использовать функции glibc удобнее, но, вызывая непосредственно системные вызовы, например: `open()`, `read()`, `write()`, `close()`, можно немного повысить производительность приложения — ведь вы минуете glibc, которая все равно использует те же системные вызовы.

VFS — это *виртуальная файловая система*. Именно она позволяет добиться существующего сейчас уровня абстракции. Каждая файловая система имеет свои осо-

бенности. Если бы не было VFS, то пришлось бы разрабатывать разные версии системных вызовов для каждого типа поддерживаемой файловой системы, например `open_ext2()` для открытия файла, находящегося на файловой системе ext2, или `open_vfat()` — для VFAT. Другими словами, VFS делает системные вызовы независимыми от типа используемой файловой системы.

Драйверы устройств используются для физического доступа к носителям данных. Ведь эти самые носители тоже различны — в компьютере может быть установлено несколько жестких дисков с разными интерфейсами, например диски PATA и SATA.

Схематически раздел диска с файловой системой ext2/3 можно представить, как показано на рис. 14.2.

Жесткий диск физически разбивается на секторы по 512 байтов каждый. Первый сектор каждого раздела является *загрузочной областью*. В загрузочной области первичного раздела находится *главная загрузочная запись* (Master Boot Record, MBR) — программа, которая запускает операционную систему. На других разделах такой записи нет.

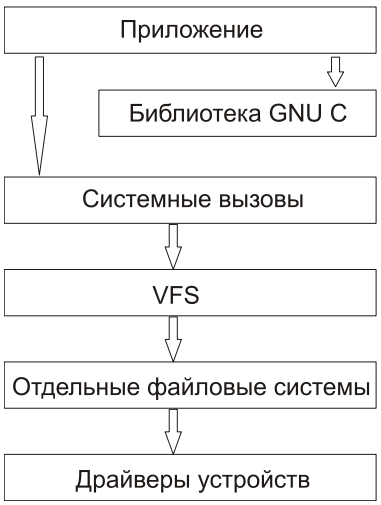


Рис. 14.1. Архитектура файловой системы

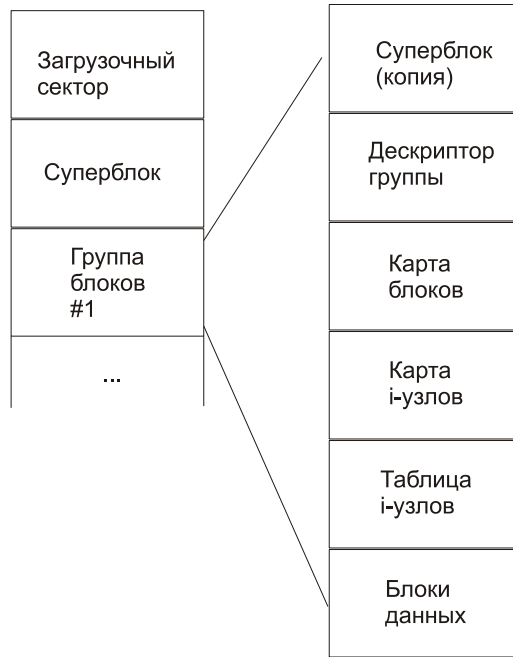


Рис. 14.2. Структура файловой системы

Все последующие (после загрузочной записи) секторы объединены в логические блоки. Блок — это наименьшая адресуемая порция данных. Размер блока может быть 1, 2 или 4 Кбайт. Блоки группируются в группы блоков. Нумерация групп начинается с 1.

После загрузочного сектора следует *суперблок*, хранящий всю информацию о файловой системе. Размер суперблока — 1 Кбайт (1024 байта). Суперблок дублируется в каждой группе блоков, что позволяет восстановить его в случае повреждения файловой системы.

Структура суперблока описана в файле /usr/src/linux/include/linux/fs.h:

```
struct super_block {
       struct_head s_list;   // Двусвязный список всех смонтированных ФС
       unsigned long s_blocksize;
       struct file_system_type *s_type;
       struct super_operations *s_op;
       struct semaphore      s_lock;
       int s_need_sync_fs;
...
}
```

В группе блоков после копии суперблока следует дескриптор группы блоков, который хранит информацию о физических координатах карт блоков и i-узлов, а также таблицы i-узлов (здесь i-узел — inode, информационный узел).

Карта блоков (block map) содержит информацию об используемых блоках и служит для поиска свободных блоков при выделении места для файла. Каждому файлу соответствует только один i-узел, хранящий метаданные файла — все атрибуты файла, кроме его имени. Карта i-узлов следует сразу после карты блоков. С помощью карты i-узлов можно определить, какой i-узел используется, а какой — занят.

Кроме атрибутов файла в i-узле хранится указатель на данные файла. Обычно это массив из 15 адресов блоков, 12 из которых непосредственно ссылаются на номера блоков, хранящие данные файла. Если данные занимают более 12 блоков (напомню, что обычно 1 блок = 1 Кбайт), то используется косвенная адресация. Поэтому следующий адрес (13-й) — это адрес блока, где находится список адресов других блоков, содержащих данные файла.

Не нужно быть гением в математике, чтобы вычислить, сколько блоков можно разместить путем косвенной адресации. Все зависит от размера блока, который может быть 1, 2 или 4 Кбайт. Следовательно, можно адресовать 256, 512 или 1024 блока. А что делать, если файл еще больше? Тогда используется двойная и тройная косвенная адресации. 14-й адрес — это адрес блока, содержащего список последующих адресов блоков данных этого файла, 15-й адрес используются тройной косвенной адресацией и содержит список адресов блоков, которые являются блоками двойной косвенной адресации.

Одним словом, максимальный размер файла может быть очень большим. Для ext3 — это 1 Тбайт. Хотя ядро 2.6 поддерживает блочные устройства размером 16 Тбайт, максимальный размер файловой системы — "всего" 4 Тбайт. Одно радует — можно создать образ двухслойного и двухстороннего DVD-диска (примерно 18 Гбайт).

Ранее было сказано, что в i-узле хранится вся информация о файле, кроме его имени. Имя файла хранится в каталоге, к которому принадлежит файл. А отсюда сле-

дует, что одному i-узлу может соответствовать неограниченное количество имен файла (ссылок). При этом ссылки (дополнительные имена) могут находиться как в одном каталоге с исходным файлом, так и в любом другом каталоге файловой системы.

Как мы уже знаем, в Linux есть обычные файлы и есть файлы устройств. В чем между ними разница? Эта разница проявляется на уровне i-узла — i-узел обычного файла указывает на блоки данных, а i-узел файла устройства указывает на адрес драйвера в ядре Linux.

14.4. Монтирование файловых систем

14.4.1. Команды *mount* и *umount*

Чтобы работать с какой-либо файловой системой, необходимо *примонтировать* ее к корневой файловой системе. Например, вставив в дисковод дискету, нужно подмонтировать файловую систему дискеты к корневой файловой системе — только так мы сможем получить доступ к файлам и каталогам, которые на этой дискете записаны. Аналогичная ситуация с жесткими, оптическими дисками и другими носителями данных.

Если вы хотите заменить сменный носитель данных (дискету, компакт-диск), вам нужно сначала размонтировать файловую систему, затем извлечь носитель данных, установить новый и заново смонтировать файловую систему. В случае с дискетой о размонтировании должны помнить вы сами, поскольку при этом выполняется синхронизация буферов ввода/вывода и файловой системы, т. е. данные физически записываются на диск, если это еще не было сделано. А компакт-диск система не разрешит вам извлечь, если он не размонтирован. В свою очередь, размонтировать файловую систему можно, только когда ни один процесс ее не использует.

При завершении работы системы (перезагрузке, выключении компьютера) размонтирование всех файловых систем выполняется автоматически.

Команда монтирования (ее нужно выполнять с привилегиями root) выглядит так:

```
# mount [опции] <устройство> <точка монтирования>
```

Точка монтирования — это каталог, через который будет осуществляться доступ к монтируемой файловой системе. Например, если вы подмонтировали компакт-диск к каталогу /mnt/cdrom, то получить доступ к файлам и каталогам, записанным на компакт-диске, можно будет через точку монтирования (именно этот каталог /mnt/cdrom). Точкой монтирования может быть любой каталог корневой файловой системы, хоть /aaa-111. Главное, чтобы этот каталог существовал на момент монтирования файловой системы.

В некоторых современных дистрибутивах запрещен вход в систему под именем суперпользователя — root. Поэтому для выполнения команд с привилегиями root вам нужно использовать команду `sudo`. Например, чтобы выполнить команду монтирования привода компакт-диска, вам нужно ввести команду:

```
sudo mount /dev/hdc /mnt/cdrom
```

Перед выполнением команды `mount` команда `sudo` попросит вас ввести пароль root. Если введенный пароль правильный, то будет выполнена команда `mount`.

Для размонтирования файловой системы используется команда `umount`:

```
# umount <устройство или точка монтирования>
```

14.4.2. Файлы устройств и монтирование

В этой главе мы уже говорили о файлах устройств. Здесь мы вернемся к ним снова, но в контексте монтирования файловой системы.

Как уже было отмечено, для Linux нет разницы между устройством и файлом. Все устройства системы представлены в корневой файловой системе как обычные файлы. Например, /dev/fd0 — это ваш дисковод для гибких дисков, /dev/hda (/dev/sda) — жесткий диск. Файлы устройств хранятся в каталоге /dev.

Жесткие диски

С жесткими дисками сложнее всего, поскольку одно и то же устройство может в разных версиях одного и того же дистрибутива называться по-разному. Например, мой IDE-диск, подключенный как первичный мастер, в Fedora 5 все еще назывался /dev/hda, а в Fedora 8 он стал называться /dev/sda. Раньше накопители, подключающиеся к интерфейсу IDE (PATA), назывались /dev/hd*x*, а SCSI/SATA-накопители — /dev/sd*x* (где в обоих случаях *x* — буква устройства).

После принятия udev и глобального уникального идентификатора устройств (UUID) все дисковые устройства, вне зависимости от интерфейса подключения (PATA, SATA, SCSI), называются /dev/sd*x*, где *x* — буква устройства. Все современные дистрибутивы поддерживают udev и UUID. Так что не удивляйтесь, если вдруг ваш старенький IDE-винчестер будет назван /dev/sda. С одной стороны, это вносит некоторую путаницу *(см. разд. 14.4.5)*. С другой стороны, все современные компьютеры оснащены именно SATA-дисками (т. к. PATA-диски уже устарели, а SCSI — дорогие), а на современных материнских платах только один контроллер IDE (PATA), потому многие пользователи даже ничего не заметят.

> ***Пояснение***
>
> udev — это менеджер устройств, используемый в ядрах Linux версии 2.6. Пришел на смену более громоздкой псевдофайловой системе devfs. Управляет всеми манипуляциями с файлами из каталога /dev.

Рассмотрим ситуацию с жесткими дисками чуть подробнее. Пусть у нас есть устройство /dev/sda. На жестком диске, понятное дело, может быть несколько разделов (логических дисков). В нашем случае на диске имеются три раздела, которые в Windows называются C:, D: и E:. Диск C: обычно является загрузочным (активным), поэтому он будет записан в самом начале диска. Нумерация разделов жесткого диска в Linux начинается с 1, поэтому в большинстве случаев диску C: будет соответствовать имя /dev/sda1 — первый раздел на первом жестком диске.

Резонно предположить, что двум оставшимся разделам (D: и E:) были присвоены имена /dev/sda2 и /dev/sda3. Это может быть так — и не так. Сейчас поясню. Раздел

может быть первичным (primary partition), расширенным (extended partition) или логическим (logical partition). Всего на диске может быть или четыре первичных раздела, или три первичных и один расширенный.

Пусть на жестком диске есть четыре первичных раздела, для которых зарезервированы номера 1, 2, 3, 4. Если разделы D: и E: — первичные, то им будут присвоены имена /dev/sda2 и /dev/sda3. Но в большинстве случаев данные разделы являются логическими, а логические разделы содержатся в расширенном разделе (там может быть максимум 11 логических разделов). При этом в Windows расширенному разделу не присваивается буква, потому что этот раздел не содержит данных пользователя, а только информацию о логических разделах. Логические разделы именуются, начиная с 5, т. е. если разделы D: и E: — логические, то им будут присвоены имена /dev/sda5 и /dev/sda6 соответственно.

Узнать номер раздела очень просто: достаточно запустить утилиту, работающую с таблицей разделов диска — `fdisk`.

Чтобы узнать номера разделов первого жесткого диска (/dev/hda), введите команду:

```
# /sbin/fdisk /dev/sda
```

После этого вы увидите приглашение `fdisk`. В ответ на приглашение нужно ввести `p` и нажать клавишу <Enter>. Вы увидите таблицу разделов (рис. 14.3). После этого для выхода из программы введите `q` и нажмите клавишу <Enter>.

```
1) программами, запускаемым при загрузке (напр., старые версии LILO)
2) загрузкой и программами разметки из других ОС
   (напр., DOS FDISK, OS/2 FDISK)

Команда (m для справки): p

Диск /dev/sda: 160.0 ГБ, 160041885696 байт
255 heads, 63 sectors/track, 19457 cylinders
Units = цилиндры of 16065 * 512 = 8225280 bytes
Disk identifier: 0xe905e905

Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/sda1   *           1         543     4361616    b  W95 FAT32
/dev/sda2             544       19457   151926705    f  W95 расшир. (LBA)
/dev/sda5             544        1021     3839503+  83  Linux
/dev/sda6            1022        1759     5927953+  83  Linux
/dev/sda7            1760        1825      530113+  82  Linux своп / Solaris
/dev/sda8            1826        5963    33238453+   b  W95 FAT32
/dev/sda9            5964       10101    33238453+   b  W95 FAT32
/dev/sda10          10102       14268    33471396    b  W95 FAT32
/dev/sda11          14269       16949    21535101    b  W95 FAT32
/dev/sda12          16950       19457    20145478+   b  W95 FAT32

Команда (m для справки):
```

Рис. 14.3. Таблица разделов жесткого диска

На рис. 14.3 изображена таблица разделов моего первого жесткого диска. Первый раздел (это мой диск C:, где установлена система Windows) — первичный. Сразу после него расположен расширенный раздел (его номер — 2). Следующий за

ним — логический раздел (номер 5). Разделы с номерами 3 и 4 пропущены, потому что их нет на моем жестком диске. Это те самые первичные разделы, которые я не создал — они мне не нужны.

Приводы оптических дисков

Файл устройства для чтения CD- или DVD-дисков называется /dev/sr*N* (или /dev/scd*N*), где *N* — номер устройства. Если компьютер оборудован всего одним оптическим устройством, то оно будет называться /dev/sr0 (или /dev/scd0). Когда система видит, что устройство является приводом CD-ROM, то автоматически создается ссылка /dev/cdrom. А если ваш привод умеет читать и DVD-диски, то появится еще одна ссылка — /dev/dvd. Монтирование привода для чтения оптических дисков осуществляется вводом одной из трех команд:

```
sudo mount /dev/sr0 /mnt/cdrom
sudo mount /dev/cdrom /mnt/cdrom
sudo mount /dev/dvd /mnt/cdrom
```

После этого обратиться к файлам, записанным на диске, можно будет через каталог /mnt/cdrom. Напомню, что этот каталог должен существовать.

Дискеты

Аналогичная ситуация и с дискетами. В системе может быть установлено два дисковода для дискет: первый (/dev/fd0) и второй (/dev/fd1). Для их монтирования можно задать команды:

```
sudo mount /dev/fd0 /mnt/floppy
sudo mount /dev/fd1 /mnt/floppy
```

Напомню, что в Windows-терминологии устройство /dev/fd0 — это диск A:, а устройство /dev/fd1 — диск B:.

Флешки и USB-диски

Флешки и USB-диски в системе отображаются как обычные жесткие диски, имена у них такие же (/dev/sd?). Например, если у вас всего один жесткий диск (его имя /dev/sda) и вы подключили к компьютеру флешку, то ее имя будет /dev/sdb. Подмонтировать флешку (вот только зачем, не забываем об автоматическом монтировании) можно так:

```
sudo mount /dev/sdb /mnt/usb
```

Перед физическим отключением флешки и USB-винчестера (особенно винчестера, учитывая его физическое устройство) нужно щелкнуть на значке флешки на рабочем столе и выбрать команду **Безопасно отключить носитель** (или **Отсоединить том**, **Извлечь** — название команды в графическом интерфейсе зависит от дистрибутива Linux).

14.4.3. Опции монтирования файловых систем

Теперь, когда мы знаем номер раздела, можно подмонтировать его файловую систему. Делается это так:

```
# mount <раздел> <точка монтирования>
```

Например:

```
# mount /dev/sda5 /mnt/win_d
```

У команды `mount` довольно много опций, но на практике наиболее часто используются только некоторые из них: `-t`, `-r`, `-w`, `-a`.

- Параметр `-t` позволяет задать тип файловой системы. Обычно программа сама определяет файловую систему, но иногда это у нее не получается. Тогда мы должны ей помочь. Формат использования этого параметра следующий:

  ```
  # mount -t <файловая система> <устройство> <точка монтирования>
  ```

  Например:

  ```
  # mount -t iso9660 /dev/hdc /mnt/cdrom
  ```

  Вот опции для указания наиболее популярных монтируемых файловых систем:

  - `ext2` или `ext3` — файловая система Linux;
  - `iso9660` — указывается при монтировании CD-ROM;
  - `vfat` — FAT, FAT32 (поддерживается Windows 9*x*, ME, XP);
  - `ntfs` — NT File System (поддерживается Windows NT, XP), будет использована стандартная поддержка NTFS, при которой NTFS-раздел доступен только для чтения;
  - `ntfs-3g` — будет использован модуль ntfs-3g, входящий в большинство современных дистрибутивов. Данный модуль позволяет производить запись информации на NTFS-разделы.

  Примечание

  Если в вашем дистрибутиве нет модуля ntfs-3g, т. е. при попытке указания данной файловой системы вы увидели сообщение об ошибке, тогда вы можете скачать его с сайта **www.ntfs-3g.org**. На данном сайте доступны как исходные коды, так и уже скомпилированные для разных дистрибутивов пакеты.

- Параметр `-r` монтирует указанную файловую систему в режиме "только чтение".
- Параметр `-w` монтирует файловую систему в режиме "чтение/запись". Данный параметр используется по умолчанию для файловых систем, поддерживающих запись (например, NTFS по умолчанию запись не поддерживает, как и файловые системы CD/DVD-дисков).
- Параметр `-a` используется для монтирования всех файловых систем, указанных в файле /etc/fstab (кроме тех, для которых указано `noauto` — такие файловые системы нужно монтировать вручную). При загрузке системы вызывается команда `mount` с параметром `-a`.

Если вы не можете смонтировать NTFS-раздел с помощью опции `ntfs-3g`, то, вероятнее всего, он был неправильно размонтирован (например, работа Windows не была завершена корректно). В этом случае для монтирования раздела нужно использовать опцию `-o force`, например:

```
sudo mount -t ntfs-3g /dev/sdb1 /media/usb -o force
```

14.4.4. Монтирование разделов при загрузке

Если вы не хотите при каждой загрузке монтировать постоянные файловые системы (например, ваши Windows-разделы), то вам нужно прописать их в файле /etc/fstab. Обратите внимание: в этом файле не нужно прописывать файловые системы сменных носителей (дисковода, CD/DVD-привода, Flash-диска). Следует отметить, что программы установки некоторых дистрибутивов, например Mandriva, читают таблицу разделов и автоматически заполняют файл /etc/fstab. В результате все ваши Windows-разделы доступны сразу после установки системы. К сожалению, не все дистрибутивы могут похвастаться такой интеллектуальностью, поэтому вам нужно знать формат файла fstab:

```
устройство точка_монтирования тип_ФС опции флаг_РК флаг_проверки
```

Здесь: `тип_ФС` — это тип файловой системы, а `флаг_РК` — флаг резервного копирования. Если он установлен (`1`), то программа `dump` заархивирует данную файловую систему при создании резервной копии. Если не установлен (`0`), то резервная копия этой файловой системы создаваться не будет. `Флаг проверки` устанавливает, будет ли данная файловая система проверяться на наличие ошибок программой `fsck`. Проверка производится в двух случаях:

- если файловая система размонтирована некорректно;
- если достигнуто максимальное число операций монтирования для этой файловой системы.

Поле опций содержит важные параметры файловой системы. Некоторые из них представлены в табл. 14.1.

***Таблица 14.1.** Опции монтирования файловой системы в файле /etc/fstab*

Опция	Описание
`auto`	Файловая система должна монтироваться автоматически при загрузке. Опция используется по умолчанию, поэтому ее указывать не обязательно
`noauto`	Файловая система не монтируется при загрузке системы (при выполнении команды `mount -a`), но ее можно смонтировать вручную с помощью все той же команды `mount`
`defaults`	Используется стандартный набор опций, установленных по умолчанию
`exec`	Разрешает запуск выполняемых файлов для данной файловой системы. Эта опция используется по умолчанию
`noexec`	Запрещает запуск выполняемых файлов для данной файловой системы

Таблица 14.1 *(окончание)*

Опция	Описание
`ro`	Монтирование в режиме "только чтение"
`rw`	Монтирование в режиме "чтение/запись". Используется по умолчанию для файловых систем, поддерживающих запись
`user`	Данную файловую систему разрешается монтировать/размонтировать обычному пользователю (не root)
`nouser`	Файловую систему может монтировать только пользователь root. Используется по умолчанию
`umask`	Определяет маску прав доступа при создании файлов. Для файловых систем не Linux'а маску нужно установить так: `umask=0`
`utf8`	Применяется только на дистрибутивах, которые используют кодировку UTF-8 в качестве кодировки локали. В старых дистрибутивах (где используется KOI8-R) для корректного отображения русских имен файлов на Windows-разделах нужно задать параметры `iocharset=koi8-u,codepage=866`

Примечание

Редактировать файл /etc/fstab, как и любой другой файл из каталога /etc, можно в любом текстовом редакторе (например, gedit, kate), но перед этим нужно получить права root (команды `su` или `sudo`).

Рассмотрим небольшой пример:

```
/dev/sdc /mnt/cdrom auto umask=0,user,noauto,ro,exec 0 0
/dev/sda1 /mnt/win_c vfat umask=0,utf8 0 0
```

Первая строка — это строка монтирования файловой системы компакт-диска, а вторая — строка монтирования диска C:.

- Начнем с первой строки. `/dev/hdc` — это имя устройства CD-ROM. Точка монтирования — `/mnt/cdrom`. Понятно, что этот каталог должен существовать. Обратите внимание: в качестве файловой системы не указывается жестко iso9660, поскольку компакт-диск может быть записан в другой файловой системе, поэтому в качестве типа файловой системы задано `auto`, т. е. автоматическое определение. Теперь идет довольно длинный набор опций. Ясно, что `umask` установлен в ноль, поскольку файловая система компакт-диска не поддерживает права доступа Linux. Параметр `user` говорит о том, что данную файловую систему можно монтировать обычному пользователю. Параметр `noauto` запрещает автоматическое монтирование этой файловой системы, что правильно — ведь на момент монтирования в приводе может и не быть компакт-диска. Опция `ro` разрешает монтирование в режиме "только чтение", а `exec` разрешает запускать исполнимые файлы. Понятно, что компакт-диск не нуждается ни в проверке, ни в создании резервной копии, поэтому два последних флага равны нулю.
- Вторая строка проще. Первые два поля — это устройство и точка монтирования. Третье — тип файловой системы. Файловая система постоянна, поэтому можно явно указать тип файловой системы (`vfat`), а не `auto`. Опция `umask`, как

и в предыдущем случае, равна нулю. Указание опции `utf8` позволяет корректно отображать русскоязычные имена файлов и каталогов.

14.4.5. Подробно о UUID и файле /etc/fstab

Пока вы еще не успели забыть формат файла /etc/fstab, нужно поговорить о UUID (Universally Unique Identifier), или о *длинных именах* дисков. В некоторых дистрибутивах, например в Ubuntu, вместо имени носителя (первое поле файла fstab) указывается его ID, поэтому fstab выглядит устрашающе, например вот так:

```
# /dev/sda6
UUID=1f049af9-2bdd-43bf-a16c-ff5859a4116a / ext3 defaults 0 1
# /dev/sda1
UUID=45AE-84D9  /media/hda1  vfat  defaults,utf8,umask=007 0 0
```

В SUSE идентификаторы устройств указываются немного иначе:

```
/dev/disk/by-id/scsi-SATA_WDC_WD1600JB-00_WD-WCANM7959048-part5 / ext3
acl,user_xattr 1 1
/dev/disk/by-id/scsi-SATA_WDC_WD1600JB-00_WD-WCANM7959048-part7 swap swap
defaults 0 0
```

Понятно, что использовать короткие имена вроде /dev/sda1 намного проще, чем идентификаторы в стиле `1f049af9-2bdd-43bf-a16c-ff5859a4116a`. Использование имен дисков еще никто не отменял, поэтому вместо идентификатора носителя можете смело указывать его файл устройства — так вам будет значительно проще!

Но все же вам нужно знать соответствие длинных имен коротким именам устройств. Ведь система использует именно эти имена, а в файле /etc/fstab не всегда указывается, какой идентификатор принадлежит какому короткому имени устройства (или указывается, но не для всех разделов).

Узнать "длинные имена" устройства можно с помощью простой команды:

```
ls -l /dev/disk/by-uuid/
```

Спрашивается, зачем были введены длинные имена, если короткие имена были удобнее, во всяком случае для пользователей? Оказывается, разработчики Linux в первую очередь и заботились как раз о пользователях. Возьмем обычный IDE-диск. Как известно, данный диск можно подключить либо к первичному (primary), либо к вторичному (secondary), если он есть, контроллеру. В зависимости от положения перемычки выбора режима винчестер может быть либо главным устройством (master), либо подчиненным (slave). Таким образом, в зависимости от контроллера, к которому подключается диск, изменяется его короткое имя — hda (primary master), hdb (primary slave), hdc (secondary master), hdd (secondary slave). То же самое происходит с SATA/SCSI-винчестерами — при изменении параметров подключения изменяется и короткое имя устройства.

При использовании же длинных имен идентификатор дискового устройства остается постоянным вне зависимости от типа подключения устройства к контроллеру. Именно поэтому длинные имена дисков часто также называются *постоянными* именами

(persistent name). Получается, что раньше вы могли ошибочно подключить жесткий диск немного иначе, и разделы, которые назывались, скажем, /dev/hda*N*, стали называться /dev/hdb*N*. Понятно, что загрузить Linux с такого диска не получится, поскольку везде указаны другие имена устройств. Если же используются длинные имена дисков, система загрузится в любом случае, как бы вы ни подключили жесткий диск. Удобно? Конечно.

Но это еще не все. Постоянные имена — это только первая причина. Вторая причина заключается в обновлении библиотеки libata. В новой версии libata все PATA-устройства именуются не как hd*x*, а как sd*x*, что (как отмечалось в этой главе ранее) вносит некую путаницу. Длинные имена дисков от этого не изменяются, поэтому они избавляют пользователя от беспокойства по поводу того, что его старый IDE-диск вдруг превратился в SATA/SCSI-диск.

При использовании UUID однозначно идентифицировать раздел диска можно несколькими способами:

- `UUID=45AE-84D9 /media/sda1 vfat defaults,utf8,umask=007, gid=46 0 0` — здесь с помощью параметра UUID указывается идентификатор диска;
- `/dev/disk/by-id/scsi-SATA_WDC_WD1600JB-00_WD-WCANM7959048-part7 swap swap defaults 0 0` — здесь указывается длинное имя устройства диска;
- `LABEL=/ / ext3 defaults 1 1` — третий, самый компактный способ, позволяющий идентифицировать устройства по их метке.

> ***Примечание***
>
> Первый способ получения длинного имени в англоязычной литературе называется by-uuid, т. е. длинное имя составляется по UUID, второй способ называется by-id, т. е. по аппаратному идентификатору устройства. Третий способ называется by-label — по метке. Просмотреть соответствие длинных имен коротким можно с помощью команд:
>
> ```
> ls -l /dev/disk/by-uuid
> ls -l /dev/disk/by-id
> ls -l /dev/disk/by-label
> ```
>
> Но есть еще и четвертый способ, который называется by-path. В этом случае имя генерируется по sysfs. Данный способ является наименее используемым, поэтому вы редко столкнетесь с ним.

Узнать метки разделов можно с помощью команды:

```
ls -lF /dev/disk/by-label
```

Установить метку можно с помощью команд, указанных в табл. 14.2.

Таблица 14.2. *Команды для установки меток разделов*

Файловая система	Команда
ext2/ext3/ext4	`# e2label /dev/XXX <метка>`
ReiserFS	`# reiserfstune -l <метка> /dev/XXX`
JFS	`# jfs_tune -L <метка> /dev/XXX`

***Таблица 14.2** (окончание)*

Файловая система	Команда
XFS	`# xfs_admin -L <label> /dev/XXX`
FAT/FAT32	Только средствами Windows
NTFS	`# ntfslabel /dev/XXX <метка>`

В файле /etc/fstab вы можете использовать длинные имена в любом формате. Можно указывать имена устройств в виде: `/dev/disk/by-uuid/*`, `/dev/disk/by-id/*` или `/dev/disk/by-label/*`, можно использовать параметры `UUID=идентификатор` или `LABEL=метка`. Используйте тот способ, который вам больше нравится.

14.4.6. Системный вызов *mount()*

Для монтирования файловой системы используется системный вызов `mount()`, для размонтирования используется системный вызов `umount()`. Чтобы `mount()` был выполнен, программа должна быть запущена с привилегиями пользователя root.

Вряд ли вы будете писать программу, использующую системный вызов `mount()`. Если, конечно, вы надумали написать свой собственный аналог команды `mount`, тогда рекомендую вам проанализировать ее исходный код, который содержится в файле mount.c. Пример кода, реализующего монтирование файловой системы (функция `do_mount`), представлен в листинге 14.1. Если кому-то интересно, то в листинге 14.1 приведен код из файла mount.c набора утилит BusyBox, который используется во всех встраиваемых UNIX-подобных операционных системах.

Листинг 14.1. Функция do_mount

```
static int
do_mount(char *specialfile, char *dir, char *filesystemtype,
                long flags, void *string_flags, int useMtab, int fakeIt,
                char *mtab_opts)
{
        int status = 0;
        char *lofile = NULL;

// Анализируем флаги, опции — все, что передано функции
#if defined BB_MTAB
        if (fakeIt == FALSE)
#endif
        {
#if defined BB_FEATURE_MOUNT_LOOP
                if (use_loop==TRUE) {
                        int loro = flags & MS_RDONLY;
                        char *lofile = specialfile;
```

```
                        specialfile = find_unused_loop_device();
                        if (specialfile == NULL) {
                                fprintf(stderr, "Could not find a spare loop
device\n");
                                return (FALSE);
                        }
                        if (set_loop(specialfile, lofile, 0, &loro)) {
                                fprintf(stderr, "Could not setup loop device\n");
                                return (FALSE);
                        }
                        if (!(flags & MS_RDONLY) && loro) { /* loop is ro, but
wanted rw */
                                fprintf(stderr, "WARNING: loop device is read-
only\n");
                                flags &= ~MS_RDONLY;
                        }
                }
#endif
// Вызываем mount()
                status = mount(specialfile, dir, filesystemtype, flags,
string_flags);
        }


        /* Если вызов успешен, т. е. status = 0, больше ничего не
        нужно делать, просто возвращаем TRUE */
        if (status == 0) {

// Если определена BB_MTAB, записываем информацию о смонтированной
// файловой системе в файл mtab
#if defined BB_MTAB
        if (useMtab == TRUE) {                write_mtab(specialfile, dir,
filesystemtype, flags, mtab_opts);
                }
#endif
                return (TRUE);
        }

        /* Вызов mount() провалился */
#if defined BB_FEATURE_MOUNT_LOOP
        if (lofile != NULL) {
                del_loop(specialfile);
        }
#endif

// Если ошибка = EPERM, скорее всего, программа не запущена
// с правами root
```

```
        if (errno == EPERM) {
                fatalError("mount: permission denied. Are you root?\n");
        }

        return (FALSE);
}
```

Исходный код команды `mount` (файл mount.c) вы найдете у себя на жестком диске. Если вы не нашли этот файл, тогда вы без проблем найдете его в Интернете. Описание системного вызова `mount()` тоже можно найти на вашем компьютере, воспользовавшись программой `man`. В Интернете описание `mount()` доступно по многочисленным адресам, один из которых приведен ниже:

http://www.opennet.ru/man.shtml?topic=mount&category=2&russian=0

Не игнорируйте эту ссылку! Ссылка указывает на русский man-файл системного вызова `mount()`, который не всегда будет в вашей системе — часто доступен только его англоязычный вариант.

В последующих главах мы рассмотрим, как работать с файлами и каталогами в Linux: будут рассмотрены не только команды Linux, но и системные вызовы, используя которые вы можете писать свои собственные программы.

ГЛАВА 15

Операции над каталогами

15.1. Команды для работы с каталогами

Сначала рассмотрим, как выполнить основные операции над каталогами с помощью команд оболочки, затем займемся программированием. Основные команды для работы с каталогами приведены в табл. 15.1.

Таблица 15.1. *Основные команды для работы с каталогами*

Команда	Описание
`mkdir <каталог>`	Создание каталога
`cd <каталог>`	Изменение каталога
`ls <каталог>`	Вывод содержимого каталога
`rmdir <каталог>`	Удаление пустого каталога
`rm -r <каталог>`	Рекурсивное удаление каталога

При указании имени каталога можно использовать следующие символы:

- `.` — означает текущий каталог, если вы введете команду `cat ./file`, то она выведет файл `file`, который находится в текущем каталоге;
- `..` — родительский каталог, например команда `cd ..`, перейдет на один уровень "вверх" по дереву файловой системы;
- `~` — домашний каталог пользователя (об этом мы поговорим позже).

Теперь рассмотрим команды для работы с файлами на практике. Введите следующие команды:

```
mkdir directory
cd directory
touch file1.txt
```

```
touch file2.txt
ls
cd ..
ls directory
rm directory
rmdir directory
rm -r directory
```

Первая команда (`mkdir`) создает каталог directory в текущем каталоге. Вторая команда (`cd`) переходит (изменяет каталог) в только что созданный каталог. Следующие две команды `touch` создают в новом каталоге два файла — file1.txt и file2.txt.

Команда `ls` без указания каталога выводит содержимое текущего каталога. Команда `cd ..` переходит в родительский каталог. Как уже было отмечено, в Linux родительский каталог обозначается как "..", а текущий как ".". То есть, находясь в каталоге directory, мы можем обращаться к файлам file1.txt и file2.txt без указания каталога или же как ./file1.txt и ./file2.txt.

Еще раз обратите внимание: в Linux, в отличие от Windows, для разделения элементов пути используется прямой слэш (/), а не обратный (\).

Кроме обозначений ".." и "." в Linux часто используется обозначение "~" — это домашний каталог. Предположим, что наш домашний каталог /home/den. В нем мы создали подкаталог dir и поместили в него файл file1.txt. Полный путь к файлу можно записать так:

```
/home/den/dir/file1.txt
```

или так:

```
~/dir/file1.txt
```

Как видите, тильда (~) заменяет часть пути. Поскольку мы находимся в родительском для каталога directory каталоге, для того чтобы вывести содержимое только что созданного каталога, в команде `ls` нам нужно четко указать имя каталога:

```
ls directory
```

Команда `rm` используется для удаления каталога. Но что мы видим: система отказывается удалять каталог! Пробуем удалить его командой `rmdir`, но и тут отказ. Система сообщает нам, что каталог не пустой, т. е. содержит файлы. Для удаления каталога нужно удалить все файлы. Конечно, делать это не сильно хочется, поэтому проще указать опцию `-r` команды `rm` для рекурсивного удаления каталога. В этом случае сначала будут удалены все подкаталоги (и все файлы в этих подкаталогах), а затем будет удален сам каталог (рис. 15.1).

Команды `cp` и `mv` работают аналогично: для копирования (перемещения/переименования) сначала указывается каталог-источник, а потом каталог-назначение. Для каталогов желательно указывать параметр `-r`, чтобы копирование (перемещение) производилось рекурсивно.

```
[root@localhost ~]# mkdir directory
[root@localhost ~]# cd directory
[root@localhost directory]# touch file.txt
[root@localhost directory]# touch file2.txt
[root@localhost directory]# ls
file2.txt  file.txt
[root@localhost directory]# cd ..
[root@localhost ~]# ls directory
file2.txt  file.txt
[root@localhost ~]# rm directory
rm: невозможно удалить каталог `directory': Is a directory
[root@localhost ~]# rmdir directory
rmdir: `directory': Directory not empty
[root@localhost ~]# rm -r directory
rm: спуститься в каталог `directory'? y
rm: удалить пустой обычный файл `directory/file.txt'? y
rm: удалить пустой обычный файл `directory/file2.txt'? y
rm: удалить Каталог `directory'? y
[root@localhost ~]#
```

Рис. 15.1. Операции с каталогами

15.2. Функции для работы с каталогами

15.2.1. Изменение текущего каталога

В заголовочном файле unistd.h объявлено два системных вызова:

```
int chdir (const char * path);
int fchdir (int fd);
```

Функции `chdir()` нужно передать имя каталога, который после вызова `chdir()` станет текущим (рабочим) каталогом для вашей программы. Функция `fchdir()` вместо имени каталога принимает файловый дескриптор каталога (в Linux с помощью системного вызова `open()` вы можете открыть каталог как обычный файл). Обе функции возвращают –1 в случае ошибки (если не удалось изменить каталог).

15.2.2. Открываем, читаем и закрываем каталог

В *главе 8* мы познакомились с абстракцией файла — FILE. В стандартной библиотеке C для каталога создана аналогичная абстракция — DIR. Для работы с каталогами намного проще использовать именно функции стандартной библиотеки, чем аналогичные системные вызовы.

В заголовочном файле dirent.h объявлены две функции, использующиеся для открытия и закрытия каталога:

```
DIR * opendir (const char * name);
int closedir (DIR * dirp);
```

Функция `opendir()` открывает каталог с указанным именем и возвращает указатель типа DIR. По аналогии с файлами, данный указатель можно использовать для дальнейшей работы с каталогом. В случае ошибки будет возвращено значение `NULL`.

Вторая функция закрывает каталог и в случае успеха возвращает 0. В случае ошибки будет возвращено значение –1.

Аналогично функции `fchdir()` описана функция `fopendir()`, которая тоже открывает каталог, но для этого ей нужно передать файловый дескриптор, а не имя каталога:

```
DIR * fopendir (int fd);
```

После того как каталог открыт, его можно прочитать с помощью функции `readdir()`:

```
struct dirent * readdir (DIR * dirp);
```

Функция `readdir()` возвращает следующий элемент каталога в качестве структуры типа `dirent`. Чтобы просмотреть весь каталог, нужно вызвать `readdir()` столько раз, сколько элементов в каталоге. Когда каталог полностью прочитан, функция возвращает `readdir()`.

Структура `dirent` описана в заголовочном файле dirent.h так:

```
struct dirent {
  ino_t          d_ino;       /* Номер inode */
  off_t          d_off;       /* Смещение на следующий элемент */
  unsigned short d_reclen;    /* Длина этой записи */
  unsigned char  d_type;      /* Тип файла; поддерживается не
                                 всеми файловыми системами */
  char           d_name[256]; /* Имя файла */
};
```

Нас больше всего интересует элемент `d_name`, содержащий имя файла/или каталога. Сейчас мы напишем небольшую программу, демонстрирующую вывод содержимого каталога (листинг 15.1.)

Листинг 15.1. Вывод содержимого каталога (программа dir.c)

```
#include <stdio.h>
#include <dirent.h>

int main (void) {
    DIR *dir;
    struct dirent *ent;

    /* Текущий каталог */
    char directory[255] = "./";
    /* Открываем каталог */
    dir = opendir(directory);

    /* Читаем каталог и выводим имена файлов и подкаталогов */
    while ((ent=readdir(dir)) != false) {
            printf("%s\n", ent->d_name);
    }
```

```
    /* Закрываем каталог */
    closedir(dir);
    return 0;
}
```

Иногда бывает нужно повторно прочитать каталог. Чтобы не закрывать и не открывать его заново, нужно использовать функцию `rewinddir()`, которая "перематывает" указатель элемента каталога на начало:

```
void rewinddir (DIR * dirp);
```

15.2.3. Получение информации о файлах

Посмотрите на структуру `dirent`: в ней нет ничего интересного, кроме имени элемента каталога — d_name. Чтобы получить информацию о файлах, нужно использовать системные вызовы `stat()`, `fstat()` и `lstat()`:

```
int stat (const char * path, struct stat * buf);
int fstat (int fd, struct stat * buf);
int lstat (const char * path, struct stat * buf);
```

Все эти системные вызовы объявлены в заголовочном файле sys/stat.h. Системному вызову нужно передать имя файла — параметр `path`, второй параметр — структура типа `stat`, в которую будут записаны сведения о файле.

Второй системный вызов подобен первому, но вместо имени файла нужно передать файловый дескриптор `fd`. Информация о файле будет записана в ту же структуру типа `stat`.

Системный вызов `lstat()` удобно использовать для ссылки. Если файл, имя которого указано в параметре `path`, является ссылкой, то в структуру типа `stat` будет записана информация *о ссылке*, а не о файле, на который она ссылается.

Рассмотрим поля структуры `stat`:

```
struct stat {
dev_t st_dev;         /* Устройство, где расположен файл */
ino_t st_ino;         /* Номер индексного узла файла */
mode_t st_mode;       /* Права доступа к файлу */
nlink_t st_nlink;     /* К-во жестких ссылок */
uid_t st_uid;         /* UID владельца */
gid_t st_gid;         /* GID группы владельца */
dev_t st_rdev;              /* Тип устройства для
                            специальных файлов устройств */
off_t st_size;              /* Размер файла в байтах */
unsigbed long st_blksize; /* Размер блока файловой системы */
unsigned long st_blocks;    /* Число выделенных под файл блоков */
time_t st_atime;            /* Время последнего доступа к файлу */
time_t st_mtime;            /* Время последнего изменения файла */
time_t st_ctime;            /* Время создания файла */
}
```

Напишем небольшую программу, демонстрирующую получение информации о файле (листинг 15.2).

Листинг 15.2. Программа fileinfo.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>

int main (void)
{
struct stat s;

if (stat("./fileinfo.c", &s) == -1)
{
       printf("Error\n");
       return 1;
}
printf("Filename: fileinfo.c\n");
printf("UID: %d GID: %d\n", (int) s.st_uid, s.st_gid);
printf("Size: %ld\n", (long int) s.st_size);
printf("Last access time: %s", ctime(&s.st_atime));
printf("Time of last modification: %s", ctime(&s.st_mtime));

return 0;
}
```

Программа выводит информацию о файле fileinfo.c. Предполагается, что так называется эта программа, поэтому данный файл будет существовать. Во всяком случае, позаботьтесь, чтобы он существовал, или же измените имя файла.

Выводится информация об идентификаторах владельца и группы владельца, выводится размер файла и время последнего доступа/модификации файла.

Наша программа не выводит информацию о правах доступа к файлу. Подробно о правах доступа мы поговорим в *главе 18*, а пока лишь разберемся, как получить эту информацию.

Информация о правах доступа хранится в поле `st_mode` структуры `stat`. В этом же поле хранится информация и о типе файла. О правах доступа, как уже отмечалось, мы поговорим в *главе 18*, а пока напишем программу, выводящую тип файла. Для этого используются макросы, описанные в табл. 15.2.

***Таблица 15.2.** Макросы для определения типа файла*

Макрос	Возвращает `true`, если файл является
`S_ISDIR()`	каталогом
`S_ISCHR()`	символьным устройством

Таблица 15.2 *(окончание)*

Макрос	Возвращает `true`, если файл является
`S_ISBLK()`	блочным устройством
`S_ISREG()`	самым обычным файлом
`S_ISFIFO()`	именованным каналом FIFO
`S_ISLNK()`	ссылкой
`S_ISSOCK()`	сокетом

Рассмотрим небольшой пример:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
...
struct stat s;
lstat("./fileinfo.c", &s);

if (S_ISREG(s.st_mode)) printf("Обычный файл\n");
else if (S_ISDIR(s.st_mode)) printf("Каталог\n");
else if (S_ISLNK(s.st_mode)) printf("Ссылка\n");
...
```

Аналогично, можно использовать и другие макросы, представленные в табл. 15.2. Обратите внимание: в предыдущем примере мы использовали системный вызов `lstat()`, иначе макрос `S_ISLNK()` не будет работать корректно.

15.2.4. Создание и удаление каталога

В заголовочном файле unistd.h описан системный вызов для удаления каталога:

```
int rmdir(const char *pathname);
```

А вот в sys/stat.h описан системный вызов для создания каталога:

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

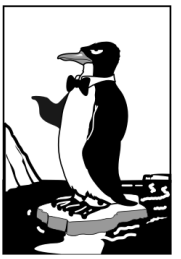
Начнем с удаления каталога. Единственный параметр — имя каталога, который вы собираетесь удалить. Перед удалением каталог должен быть пуст.

Системный вызов `mkdir()` создает каталог, имя которого указано в качестве первого параметра функции, второй параметр задает права доступа:

```
mkdir("directory", 0777);
```

Оба системных вызова возвращают `-1` в случае ошибки и `0` в случае успешного завершения операции.

ГЛАВА 16



Операции с файлами

16.1. Команды для работы с файлами

Здесь мы рассмотрим основные команды оболочки для работы с файлами в Linux (табл. 16.1), а в последующих разделах этой главы — команды для работы с каталогами, ссылками и поговорим о правах доступа к файлам и каталогам.

Таблица 16.1. *Основные команды Linux, предназначенные для работы с файлами*

Команда	Назначение
`touch <файл>`	Создает пустой файл
`cat <файл>`	Просмотр текстового файла
`tac <файла>`	Вывод содержимого текстового файла в обратном порядке, т. е. сначала выводится последняя строка, потом предпоследняя и т. д.
`cp <файл1> <файл2>`	Копирует файл `<файл1>` в файл `<файл2>`. Если `<файл2>` существует, программа попросит разрешение на его перезапись
`mv <файл1> <файл2>`	Перемещает файл `<файл1>` в файл `<файл2>`. Эту же команду можно использовать и для переименования файла
`rm <файл>`	Удаляет файл
`locate <файл>`	Производит быстрый поиск файла
`which <программа>`	Выводит каталог, в котором находится программа, если она вообще установлена. Поиск производится в каталогах, указанных в переменной окружения `PATH` (это путь поиска программ)
`less <файл>`	Используется для удобного просмотра файла с возможностью скроллинга (постраничной прокрутки)

Рассмотрим небольшую серию команд (протокол выполнения этих команд приведен на рис. 16.1):

```
touch file.txt
echo "some text" > file.txt
cat file.txt
cp file.txt file-copy.txt
cat file-copy.txt
rm file.txt
cat file.txt
mv file-copy.txt file.txt
cat file.txt
```

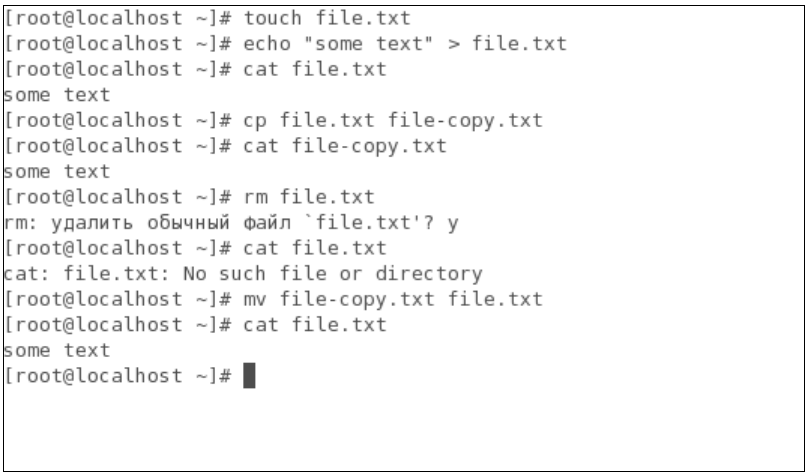


Рис. 16.1. Операции с файлом

Первая команда (`touch`) создает в текущем каталоге файл `file.txt`. Вторая команда (`echo`) записывает строку `some text` в этот же файл. Обратите внимание на символ `>` — это символ перенаправления ввода/вывода, о котором мы поговорим чуть позже.

Третья команда (`cat`) выводит содержимое файла — в файле записанная нами строка `some text`. Четвертая команда (`cp`) копирует файл `file.txt` в файл с именем `file-copy.txt`. После этого мы опять используем команду `cat`, чтобы вывести содержимое файла `file-copy.txt` — надо же убедиться, что файл действительно скопировался.

Шестая команда (`rm`) удаляет файл `file.txt`. При удалении система спрашивает, хотите ли вы удалить файл. Если хотите удалить, то нужно нажать клавишу <Y>, а если нет, то <N>. Точно ли файл удален? Убедимся в этом: введите команду `cat file.txt`. Система нам сообщает, что нет такого файла.

Восьмая команда (`mv`) переименовывает файл `file-copy.txt` в файл `file.txt`. Последняя команда выводит исходный файл `file.txt`. Думаю, особых проблем с этими командами у вас не возникло, тем более что принцип действия этих команд вам должен быть знаком по командам DOS, которые вы как квалифицированный пользователь Windows должны знать наизусть.

Вместо имени файла иногда очень удобно указать *маску имени файла*. Например, у нас есть много временных файлов, имена которых заканчиваются фрагментом tmp. Для их удаления нужно воспользоваться командой: `rm *tmp`.

Если же требуется удалить все файлы в текущем каталоге, можно просто указать звездочку: `rm *`.

Аналогично, можно использовать символ `?`, который, в отличие от звездочки, заменяющей последовательность символов произвольной длины, заменяет всего один символ. Например, нам нужно удалить все файлы, имена которых состоят из трех букв и начинаются на s:

```
rm s??
```

Будут удалены файлы s14, sqm, sr6 и т. д., но не будут тронуты файлы, имена которых состоят более чем из трех букв и которые не начинаются на s.

Маски имен можно также использовать и при работе с каталогами.

В Linux допускается, чтобы один и тот же файл существовал в системе под разными именами. Для этого используются ссылки. Ссылки бывают двух типов: жесткие и символические. Жесткие ссылки жестко привязываются к файлу — вы не можете удалить файл, пока на него указывает хотя бы одна жесткая ссылка. А вот если на файл указывают символические ссылки, его удалению ничто не помешает.

Жесткие ссылки не могут указывать на файл, который находится за пределами файловой системы. Предположим, у вас два Linux-раздела: один корневой, а второй используется для домашних файлов пользователей и монтируется к каталогу /home корневой файловой системы. Так вот, вы не можете создать в корневой файловой системе ссылку, которая ссылается на файл в файловой системе, подмонтированной к каталогу /home. Это очень важная особенность жестких ссылок. Если вам нужно создать ссылку на файл, который находится за пределами файловой системы, вам следует использовать символические ссылки.

Для создания ссылок используется команда `ln`:

```
ln file.txt link1
ln -s file.txt link2
```

Первая команда создает жесткую ссылку `link1`, ссылающуюся на текстовый файл `file.txt`. Вторая команда создает символическую ссылку `link2`, которая ссылается на этот же текстовый файл `file.txt`.

Модифицируя ссылку (все равно какую — `link1` или `link2`), вы автоматически модифицируете исходный файл — `file.txt`.

Особого внимания заслуживает операция удаления. По идее, если вы удаляете ссылку `link2`, файл `file.txt` также должен быть удален, но не тут-то было — вы не можете его удалить до тех пор, пока на него указывает хоть одна жесткая ссылка. При удалении ссылки `link2` просто будет удалена символьная ссылка, но жесткая ссылка и сам файл останутся. Если же вы удалите ссылку `link1`, будет удален и файл `file.txt`, поскольку на него больше не ссылается ни одна жесткая ссылка.

16.2. Системные вызовы для работы с файлами

16.2.1. Переименование файла: *rename()*

В заголовочном файле stdio.h объявлен системный вызов `rename()`, позволяющий переименовать файл или переместить его в рамках одной файловой системы. Рассмотрим прототип функции `rename()`:

```
int rename (const char * oldpath, const char * newpath);
```

Первый параметр — старое имя файла, второй — новое. В случае успеха функция возвращает 0, а если же что-то пошло не так, то будет возвращено значение –1.

Рассмотрим небольшой пример (листинг 16.1).

Листинг 16.1. Программа rename.c

```
#include <stdio.h>

int main (int argc, char ** argv)
{
if (argc < 3) {
       printf("Использование: rename старое_имя новое_имя\n");
       return 1;
}

if (rename(argv[1], argv[2]) == -1) {
       printf("Ошибка при переименовании файла\n");
       return 2;
}

return 0;
}
```

Программа простейшая, но вполне демонстрирует использование системного вызова `rename()`. Кстати, функция `rename()` действительно является системным вызовом, хотя и объявлена в stdio.h, где обычно объявляются стандартные библиотечные механизмы.

16.2.2. Удаление файла и каталогов: *unlink()* и *rmdir()*

Системный вызов `unlink()` объявлен в заголовочном файле unistd.h и используется для удаления файла:

```
int unlink (const char * pathname);
```

Единственный параметр — это имя удаляемого файла, в случае успеха функция возвращает 0 и –1, если файл удалить не удалось.

Ранее было отмечено, что каталоги — это также файлы. Так оно и есть, но для удаления каталога используется системный вызов `rmdir()`, который был рассмотрен в *главе 15*.

Далее приведен пример использования системного вызова `unlink()`:

```
#include <stdio.h>
#include <unistd.h>
...
if (unlink(fname) == -1) {
        printf("Не могу удалить файл\n");
        return 1;
}
```

16.2.3. Системный вызов *umask()*

Системный вызов `umask()` изменяет маску прав доступа текущего процесса:

```
mode_t umask (mode_t mask);
```

Системный вызов изменяет текущую маску прав доступа и возвращает предыдущую маску, чтобы вы могли ее сохранить и восстановить в случае необходимости. С помощью маски прав доступа вы можете задать маску для новых файлов и каталогов. Подробнее о масках прав доступа вы можете прочитать в `man umask`.

16.2.4. Работа со ссылками

В *разд. 16.1* мы познакомились со ссылками. Как уже отмечалось, ссылки бывают символическими и жесткими. Для создания жестких ссылок используется системный вызов `link()`, а для создания символических — `symlink()`:

```
int link (const char *existing, const char *new);
int symlink (const char *existing, const char *new);
```

Оба системных вызова объявлены в заголовочным файле unistd.h, обе функции возвращают –1 в случае ошибки или 0 в случае успеха.

Первый параметр (`existing`) задает имя ссылки, второй — имя файла, на который будет указывать созданная ссылка.

Чтобы узнать, на какой файл указывает ссылка, используется системный вызов `readlink()`, объявленный в unistd.h:

```
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
```

Первый параметр — имя ссылки, второй — буфер, в который будет записан путь к файлу, на который указывает ссылка, третий — размер буфера. Функция возвращает количество реально записанных в буфер символов, в случае ошибки возвращается –1.

Строки в C заканчиваются символом `'\0'`, но системный вызов `readlink()` не записывает в конец строки символ `\0`, поэтому вы должны позаботиться самостоятельно:

```
int bytes;
/* Ссылка link должна существовать, создайте ее
перед выполнением примера */
char buffer 1025;
bytes = readlink("link", buffer, 1024);
buffer[1024] = '\0';
printf("%s", buffer);
```

ГЛАВА 17

Получение информации о файловой системе

17.1. Список смонтированных файловых систем

В *главе 14* мы познакомились, наверное, со всеми особенностями файловой системы Linux. Также был рассмотрен системный вызов `mount()`, использующийся для монтирования файловых систем. После того как файловая система смонтирована, сведения о ней заносятся в файл /etc/mtab (хотя в некоторых системах для этого может использоваться другой файл, например /proc/mounts).

Заниматься чтением и анализом этого файла — это все равно, что изобретать велосипед. Формат этого файла аналогичен формату файла /etc/fstab *(см. главу 14)* и чтобы упростить труд обычного программиста, была создана функция `setmntent()`, объявленная в mntent.h:

```
FILE * setmntent (const char * FILENAME, const char * MODE);
struct mntent * getmntent (FILE * FILEP);
int endmntent (FILE * FILEP);
```

Функция `setmntent()` используется для открытия файла, в котором хранится список смонтированных файловых систем. В большинстве случаев этим файлом будет файл /etc/mtab. Это имя и нужно указать в качестве первого параметра функции. Второй параметр — режим открытия файла, принимает те же значения, что и второй аргумент функции `fopen()`.

В случае ошибки функция возвращает `NULL`, а в случае успеха — указатель на открытый файл.

Следующая функция, `getmntent()`, извлекает из открытого функцией `setmntent()` файла следующую запись типа `struct mntent`. Если `getmntent()` возвращает `NULL`, то прочитаны все записи из /etc/mtab.

Структура `mntent` выглядит так:

```
struct mntent
{
       char *mnt_fsname;     /* Имя устройства */
       char *mnt_dir;        /* Точка монтирования */
```

```
        char *mnt_type;              /* Тип файловой системы */
        char *mnt_opts;              /* Параметры монтирования */
        int mnt_freq;          /* Флаг архивации dump */
        int mnt_passno;              /* Флаг проверки fsck */
};
```

Функция `endmntent()` вызывается, когда данные о файловых системах уже прочитаны. Фактически она используется для закрытия файла /mnt/fstab.

Итак, мы научились читать записи файла /etc/mtab. Но давайте немного углубимся. Получить более подробную информацию, например размер файловой системы, количество свободных блоков, можно с помощью системного вызова `statvfs()`:

```
int statvfs (const char * PATH, struct statvfs * FS);
```

Первый параметр — это точка монтирования файловой системы, второй — структура типа `statvfs`, в которую и заносятся данные о файловой системе. Поля этой системы описаны в `man statvfs`, но нас больше всего сейчас интересуют поля `f_bsize`, `f_blocks` и `f_bavail`. Первое поле — размер блока файловой системы в байтах, второе поле — количество блоков в файловой системе. Если умножить первое поле на второе, получится размер файловой системы в байтах. Третье поле — число свободных блоков, но без учета резервных блоков (некоторые файловые системы резервируют определенное число блоков для служебных целей).

Теперь немного практики. Давайте напишем небольшую программу, выводящую информацию о смонтированных файловых системах (листинг 17.1). Результат выполнения этой программы изображен на рис. 17.1.

Листинг 17.1. Программа fsinfo.c

```
#include <stdio.h>
#include <mntent.h>
#include <sys/statvfs.h>
#include <stdlib.h>

int main(void)
{
FILE * f;
struct mntent * record;
struct statvfs fsinfo;

f = setmntent("/etc/mtab", "r");

if (f==NULL)
{
        printf("Error: can't open /etc/mtab");
        exit(1);
}
```

```
while ((record = getmntent(f)) !=NULL)
{
if (statvfs(record->mnt_dir, &fsinfo) == -1) {
        printf("Error");
        exit(1);
 }
 printf("Device: %s\n", record->mnt_fsname);
 printf("Mount point: %s\n", record->mnt_dir);
 printf("Total blocks: %ld\n", (unsigned long int) fsinfo.f_blocks);
 printf("Free blocks: %ld\n", (unsigned long int) fsinfo.f_bfree);
 printf("Block size: %ld\n", (unsigned long int) fsinfo.f_bsize);

}

endmntent(f);
return 0;
}
```

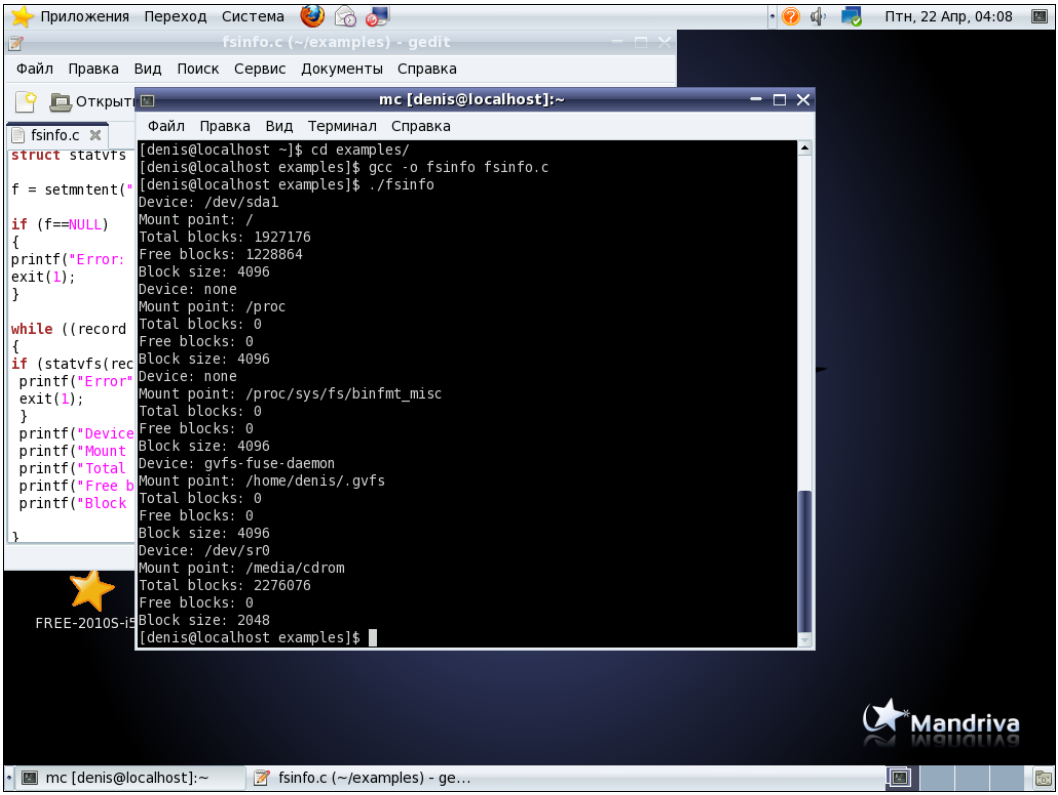


Рис. 17.1. Программа fsinfo

17.2. Функции *basename()* и *getcwd()*

При написании собственных программ бывает нужно получить имя текущего каталога. Для этого используется системный вызов `getcwd()`, объявленный в заголовочном файле unistd.h:

```
char * getcwd (char * BUFFER, size_t SIZE);
```

Небольшой пример использования `getcwd()`:

```
#include <unistd.h>
...
char * b = (char*) malloc (1024 * sizeof(char));
...
b = getcwd(b, 1024);
printf("Текущий каталог: %s\n", b);
```

Когда нужно получить только имя файла, отделив от него имя каталога, в котором он находится, используется функция `basename()`, которая объявлена в заголовочном файле strings.h:

```
char * basename (const char * PATH);
```

Пример:

```
printf("%s\n", basename("/home/den/thread.c"));
```

ГЛАВА 18

Права доступа к файлам и каталогам

18.1. Изменение прав доступа. Системный вызов *chmod()*

Для каждого каталога и файла вы можете задать права доступа. Точнее, права доступа автоматически задаются при создании каталога/файла, а вам при необходимости нужно их изменить. Какая может быть необходимость? Например, вам нужно, чтобы к вашему файлу-отчету смогли получить доступ пользователи — члены вашей группы. Или вы создали обычный текстовый файл, содержащий инструкции командного интерпретатора. Чтобы этот файл стал сценарием, вам нужно установить право на выполнение для этого файла.

Существует три права доступа: чтение (r), запись (w), выполнение (x). Для каталога право на выполнение означает право на просмотр содержимого каталога.

Вы можете установить разные права доступа для владельца (т. е. для себя), для группы владельца (т. е. для всех пользователей, входящих в одну с владельцем группу) и для прочих пользователей. Пользователь root может получить доступ к любому файлу или каталогу вне зависимости от прав, которые вы установили. Понятно, что изменить права доступа к файлу может или пользователь root или владелец файла.

Чтобы просмотреть текущие права доступа, введите команду:

```
ls -l <имя файла/каталога>
```

Например,

```
ls -l video.txt
-r--r----- 1 den group 300 Apr 11 11:11 video.txt
```

`-r--r-----` — это права доступа. Первый символ — это признак каталога. Сейчас перед нами файл. Если бы перед нами был каталог, то первый символ был бы символом `d` (от *directory*).

Последующие три символа (`r--`) определяют права доступа владельца файла или каталога. Первый символ — это чтение, второй — запись, третий — выполнение.

Как видно, владельцу разрешено только чтение этого файла, запись и выполнение запрещены, поскольку в правах доступа режимы `w` и `x` не определены.

Следующие три символа (`r--`) задают права доступа для членов группы владельца. Права такие же, как и у владельца: можно читать файл, но нельзя изменять или запускать.

Последние три символа (`---`) задают права доступа для прочих пользователей. Прочие пользователи не имеют право ни читать, ни изменять, ни выполнять файл. При попытке получить доступ к файлу они увидят сообщение "Access denied".

Примечание

После прав доступа программа `ls` выводит имя владельца файла, имя группы владельца, размер файла, дату и время создания, а также имя файла.

Права доступа задаются командой `chmod`. Существует два способа указания прав доступа: символьный (когда указываются символы, задающие право доступа — `r`, `w`, `x`) и абсолютный. Так уже заведено, что в мире UNIX чаще пользуются абсолютным методом.

Разберемся, в чем заключается этот метод. Рассмотрим следующий набор прав доступа:

```
rw-r-----
```

Данный набор прав доступа предоставляет владельцу право чтения и модификации файла (`rw-`), запускать файл владелец не может. Члены группы владельца могут только просматривать файл (`r--`), а все остальные пользователи не имеют вообще никакого доступа к файлу.

Возьмем отдельный набор прав, например для владельца:

```
rw-
```

Чтение разрешено, значит, мысленно записываем 1, запись разрешена, значит, запоминаем еще 1, а вот выполнение запрещено, поэтому запоминаем 0. Получается число 110. Если из двоичной системы перевести число 110 в восьмеричную, получится число 6. Для перевода можно воспользоваться табл. 18.1.

Таблица 18.1. *Преобразование чисел из восьмеричной системы в двоичную*

Восьмеричная система	Двоичная
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Аналогично, произведем разбор прав для членов группы владельца. Получится 100, т. е. 4. С третьим набором (---) все вообще просто — это 000, т. е. 0.

Записываем полученные числа в восьмеричной системе в порядке "владелец-группа-остальные". Получится число 640 — это и есть права доступа. Для того чтобы установить этим права доступа, выполните команду:

```
chmod 640 <имя_файла>
```

Наиболее популярные права доступа:

- 644 — владельцу можно читать и изменять файл, остальным пользователем — только читать;
- 666 — читать и изменять файл можно всем пользователям;
- 777 — всем можно читать, изменять и выполнять файл. Напомню, что для каталога право выполнения — это право просмотра оглавления каталога.

Иногда проще воспользоваться символьным методом. Например, у нас есть файл script, который нужно сделать исполнимым, для чего используется команда:

```
chmod +x script
```

Для того чтобы снять право выполнения, используется параметр `-x`:

```
chmod -x script
```

Подробнее о символьном методе вы сможете прочитать в руководстве по команде `chmod` (`man chmod`). А мы тем временем рассмотрим системный вызов `chmod()`, позволяющий изменить права доступа к файлу из вашей С-программы:

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

Первый параметр — это имя файла, права которого будут установлены в соответствии со вторым параметром, `mode`. Права доступа задаются следующими константами:

- `S_IRUSR` — право на чтение владельцем;
- `S_IWUSER` — право записи владельцем;
- `S_IXUSR` — право выполнения владельцем;
- `S_IRGRP` — право чтения группой владельца;
- `S_IWGRP` — право записи группой владельца;
- `S_IXGRP` — право выполнения группой владельца;
- `S_IROTH` — право чтения остальными пользователями;
- `S_IWOTH` — право выполнения остальными пользователями;
- `S_IXOTH` — право выполнения остальными пользователями.

Константы доступа можно комбинировать с помощью поразрядной операции `OR`. Например, для установки прав 640 (владелец может читать и изменять файл, группа

владельца — только читать, остальные — не имеют доступа к файлу) для файла file.txt можно использовать следующий системный вызов:

```
chmod ("file.txt", S_IRUSR|S_IWUSR|S_IRGRP);
```

Понятно, что для установки прав доступа программа должна быть запущена или от имени владельца файла file.txt, или от имени пользователя root.

18.2. Смена владельца файла. Системный вызов *chown()*

Если вы хотите "подарить" кому-то файл, т. е. сделать какого-то пользователя владельцем файла, то вам нужно использовать команду `chown`:

```
chown пользователь файл
```

Учтите, что, возможно, после изменения владельца файла вы сами не сможете получить к нему доступ, ведь владелец уже не вы.

Для изменения владельца файла в C-программе используется одноименный системный вызов `chown()`:

```
#include <sys/types.h>
#include <unistd.h>
int chown (const char *path, uid_t owner, gid_t group);
```

Первый параметр — это имя файла, второй параметр — идентификатор владельца (UID), третий — идентификатор группы владельца (GID). Идентификаторы можно узнать в файлах /etc/passwd (UID), /etc/group (GID), получить с помощью системных вызовов `getuid()` и `getgid()` или прочитать из командной строки (если идентификаторы передаются программе в качестве параметров) — тут все зависит от специфики вашей программы.

Опять-таки, вы можете изменить владельца файла, если сами являетесь его владельцем в данный момент (или являетесь пользователем root).

ГЛАВА 19

Псевдофайловые системы

19.1. Что такое псевдофайловая система

В Linux довольно популярны *псевдофайловые* системы. Слово "псевдо", как мы знаем, означает "почти", т. е. псевдофайловая система — не совсем файловая система в прямом смысле этого слова. Псевдофайловые системы также называются *виртуальными* файловыми системами, поскольку работают на уровне виртуальной файловой системы (Virtual File System layer).

Для большинства пользователей виртуальная файловая система выглядит как обычная файловая система — можно открыть тот или иной файл и посмотреть, что в нем записано, можно записать информацию в файл. Ради интереса зайдите в каталог /proc (это каталог псевдофайловой системы proc) и посмотрите на размер любого файла, например на размер файла /proc/filesystems. Его размер будет равен 0, как и остальных файлов этой файловой системы, но если открыть сам файл, то вы увидите, что информация в нем есть. Это объясняется тем, что содержимое файла формируется при обращении к нему, т. е. "на лету". Другими словами, виртуальная файловая система находится в оперативной памяти, а не на жестком диске. Информация попадает в файл на основании сведений, полученных от ядра.

В большинстве современных дистрибутивов используются виртуальные файловые системы sysfs и proc. Откройте файл /etc/fstab и вы увидите строки монтирования этих файловых систем:

```
sysfs     /sys       sysfs   defaults        0 0
proc      /proc      proc    defaults        0 0
```

Псевдофайловые системы sysfs и proc предоставляют информацию о системе, а также позволяют изменять важные системные параметры на лету. Настройкой системы на лету обычно занимаются администраторы серверов, которые перезагружать нежелательно, а вот программистам больше интересна другая сторона псевдофайловых систем — возможность получения информации о системе. Вы с легкостью можете написать программу, выводящую подробную информацию о системе,

но сама программа при этом будет предельно проста: она будет открывать и читать содержимое файлов из /sysfs или /proc. Не нужно изучать какие-либо системные вызовы, просто открываете файл и читаете его. Далее приведено описание виртуальных файловых систем sysfs и proc. Исследуйте файлы этих файловых систем, а идея о разработке программы родится самостоятельно — во время чтения этой главы.

19.2. Виртуальная файловая система sysfs

Виртуальная (псевдофайловая) система sysfs экспортирует в пространство пользователя информацию о ядре Linux, об имеющихся в системе устройствах и их драйверах. Впервые sysfs появилась в ядре версии 2.6. Зайдите в каталог /sys. Названия подкаталогов говорят сами за себя:

- block — содержит каталоги всех блочных устройств, имеющихся в системе в данное время (под устройством подразумевается совокупность физического устройства и его драйвера). Когда вы подключаете Flash-диск, то в любом случае в каталоге /sys/devices/ появляется новое устройство, но в каталоге /sys/block это устройство появится только при наличии соответствующих драйверов (в данном случае usb-storage);
- bus — перечень шин, поддерживаемых ядром (точнее, зарегистрированных в ядре). В каждом каталоге шины есть подкаталоги devices и drivers. В каталоге devices находятся ссылки на каталоги всех устройств, которые описаны в системе (т. е. находящихся в каталоге /sys/devices);
- class — по этому каталогу можно понять, как устройства формируются в классы. Для каждого устройства в каталоге class есть свой отдельный каталог (под устройством, как и в случае с каталогом block, подразумевается совокупность устройства и его драйвера);
- devices — содержит файлы и каталоги, которые полностью соответствуют внутреннему дереву устройств ядра;
- drivers — каталоги драйверов для загруженных устройств. Подкаталог drivers каталога шины содержит драйверы устройств, работающих на данной шине.

19.3. Виртуальная файловая система /proc

Виртуальная (псевдофайловая) система /proc — это специальный механизм, позволяющий посылать информацию ядру, модулям и процессам (кстати, потому данная файловая система так и называется — /proc, сокращение от англ. *process*). Также, используя /proc, вы можете получать информацию о процессах и изменять параметры ядра и его модулей "на лету". Для этого в /proc есть файлы, позволяющие получать информацию о системе, ядре или процессе, и есть файлы, с помощью которых можно изменять некоторые параметры системы. Первые файлы мы можем только просмотреть, а вторые — просмотреть и, если нужно, изменить.

Просмотреть информационный файл можно командой `cat`:

```
cat /proc/путь/<название_файла>
```

Записать значение в один из файлов proc можно так:

```
echo "данные" > /proc/путь/название_файла
```

19.3.1. Информационные файлы

В табл. 19.1 представлены некоторые (самые полезные) информационные proc-файлы: с их помощью вы можете получить информацию о системе.

Таблица 19.1. *Информационные proc-файлы*

Файл	Описание
/proc/version	Содержит версию ядра
/proc/cmdline	Список параметров, переданных ядру при загрузке
/proc/cpuinfo	Информация о процессоре
/proc/meminfo	Информация об использовании оперативной памяти (почти то же, что и команда `free`)
/proc/devices	Список устройств
/proc/filesystems	Файловые системы, которые поддерживаются вашей системой
/proc/mounts	Список подмонтированных файловых систем
/proc/modules	Список загруженных модулей
/proc/swaps	Список разделов и файлов подкачки, которые активны в данный момент

19.3.2. Файлы, позволяющие изменять параметры ядра

Каталог /proc/sys/kernel содержит файлы, с помощью которых можно изменять важные параметры ядра. Конечно, все файлы мы рассматривать не будем, а рассмотрим лишь те, которые используются на практике (табл. 19.2).

Таблица 19.2. *Файлы каталога /proc/sys/kernel*

Файл	Каталог
/proc/sys/kernel/ctrl-alt-del	Если данный файл содержит значение 0, то при нажатии клавиатурной комбинации <Ctrl>+<Alt>+<Del> будет выполнена так называемая мягкая перезагрузка, когда управление передается программе `init` и последняя "разгружает" систему, как при вводе команды `reboot`. Если этот файл содержит значение 1, то нажатие <Ctrl>+<Alt>+<Del> равносильно нажатию кнопки **Reset**. Сами понимаете, значение 1 устанавливать не рекомендуется
/proc/sys/kernel/domainname	Здесь находится имя домена, например **dkws.org.ua**

Таблица 19.2 *(окончание)*

Файл	Каталог
/proc/sys/kernel/hostname	Содержит имя компьютера, например den
/proc/sys/kernel/panic	При критической ошибке ядро "впадает в панику" — работа системы останавливается, а на экране красуется надпись "kernel panic" и выводится текст ошибки. Данный файл содержит значение в секундах, которое система будет ждать, пока пользователь прочитает это сообщение, после чего компьютер будет перезагружен. Значение 0 (по умолчанию) означает, что перезагружать компьютер вообще не нужно
/proc/sys/kernel/printk	Данный файл позволяет определить важность сообщения об ошибках. По умолчанию файл содержит значения 6 4 1 7. Это означает, что сообщения с уровнем приоритета 6 и ниже (чем ниже уровень, тем выше важность сообщения) будут выводиться на консоль. Для некоторых сообщений об ошибках уровень приоритета не задается. Тогда нужно установить уровень по умолчанию. Это как раз и есть второе значение — 9. Третье значение — это номер самого максимального приоритета, а последнее значение задает значение по умолчанию для первого значения. Обычно изменяют только первое значение, дабы определить, какие значения должны быть выведены на консоль, а какие — попасть в журнал демона syslog

19.3.3. Файлы, изменяющие параметры сети

В каталоге /proc/sys/net вы найдете файлы, изменяющие параметры сети (табл. 19.3).

Таблица 19.3. *Файлы каталога /proc/sys/net*

Файл	Описание
/proc/sys/net/core/message_burst	Опытные системные администраторы используют этот файл для защиты от атак на отказ (DoS). Один из примеров DoS-атаки — когда система заваливается сообщениями атакующего, а полезные сообщения системой игнорируются, потому что она не успевает реагировать на сообщения злоумышленника. В данном файле содержится значение времени (в десятых долях секунды), необходимое для принятия следующего сообщения. Значение по умолчанию — 50 (5 секунд). Сообщение, попавшее в "перерыв" (в эти 5 секунд), будет проигнорировано
/proc/sys/net/core/message_cost	Чем выше значение в этом файле, тем больше сообщений будет проигнорировано в перерыв, заданный файлом message_burst
/proc/sys/net/core/netdev_max_backlog	Задает максимальное число пакетов в очереди. По умолчанию — 300. Используется, если сетевой интерфейс передает пакеты быстрее, чем система может их обработать
/proc/sys/net/core/optmem_max	Задает максимальный размер буфера для одного сокета

19.3.4. Файлы, изменяющие параметры виртуальной памяти

В каталоге /proc/sys/vm вы найдете файлы, с помощью которых можно изменить параметры виртуальной памяти:

- в файле buffermem находятся три значения (разделяются пробелами): минимальный, средний и максимальный объем памяти, которую система может использовать для буфера. Значения по умолчанию: 2 10 60;
- в файле kswapd тоже есть три значения, которые можно использовать для управления подкачкой:
  - первое значение задает максимальное количество страниц, которые ядро будет пытаться переместить на жесткий диск за один раз;
  - второе значение — минимальное количество попыток освобождения той или иной страницы памяти;
  - третье значение задает количество страниц, которые можно записать за один раз. Значения по умолчанию 512 32 8.

19.3.5. Файлы, позволяющие изменить параметры файловых систем

Каталог /proc/sys/fs содержит файлы, изменяющие параметры файловых систем. В частности:

- файл file-max задает максимальное количество одновременно открытых файлов (по умолчанию — 4096);
- в файле inode-max содержится максимальное количество одновременно открытых индексных дескрипторов — *инодов* (максимальное значение также равно 4096);
- в файле super-max находится максимальное количество используемых суперблоков;

> ***Пояснение***
>
> Поскольку каждая файловая система имеет свой суперблок, легко догадаться, что количество подмонтируемых файловых систем не может превысить значение из файла super-max, которое по умолчанию равно 256, чего в большинстве случаев вполне достаточно. Наоборот, можно уменьшить это значение, дабы никто не мог подмонтировать больше файловых систем, чем нужно (если монтирование файловых систем разрешено обычным пользователям).

- в файле super-nr находится количество открытых суперблоков в текущий момент. Данный файл нельзя записывать, его можно только читать.

19.3.6. Как сохранить изменения

Вы узнали, как можно изменить некоторые системные параметры "на лету". Теперь вполне закономерно, что вам интересно, как сохранить изменения? Чтобы сохра-

нить измененные параметры, их нужно прописать в файле /etc/sysctl.conf (да, всего лишь нужно написать программку, выполняющую запись в файл!). Вот только формат этого файла следующий: надо отбросить /proc/sys/ в начале имени файла, а все, что останется, записать через точку, а затем через знак равенства указать значение параметра. Например, для изменения параметра /proc/sys/vm/buffermem нужно в файле /etc/sysctl.conf прописать строку:

```
vm.buffermem = 2 11 60
```

Если в вашем дистрибутиве нет файла /etc/sysctl.conf, тогда пропишите команды вида `echo "значение" > файл` в сценарий инициализации системы.

ЧАСТЬ V

Сетевое программирование

Глава 20.	Введение в TCP/IP
Глава 21.	Программирование сокетов: теория
Глава 22.	Программирование сокетов: практика

Пятая часть книги посвящена сетевому программированию в Linux. В *главе 20* будут приведены основы протокола TCP/IP. Скорее всего, вы с ними знакомы, и цель этой главы — напомнить вам то, что вы уже и так знаете. В *главе 21* — рассмотрены функции для работы с протоколом TCP/IP. А все самое интересное, а именно создание программ сервера и клиента, рассмотрено в последней главе пятой части — в *главе 22*.

ГЛАВА 20

Введение в TCP/IP

20.1. Модель OSI

В 80-х годах прошлого века появилась необходимость стандартизировать различные сетевые технологии. Ведь без стандартизации в мире компьютерных сетей воцарился бы хаос: оборудование различных производителей не смогло бы работать вместе. Поэтому международная организация по стандартизации (International Organization for Standardization, OSI) разработала *модель взаимодействия открытых систем* (Open System Interconnection, OSI). Вы также можете встретить другие названия этой модели: *семиуровневая модель OSI*, или просто *модель OSI*. Эта модель предусматривает семь уровней взаимодействия систем:

1. Физический.
2. Канальный.
3. Сетевой.
4. Транспортный.
5. Сеансовый.
6. Представительный.
7. Прикладной.

Зачем нужна была такая система? Предположим, что нам нужно заставить вместе работать две сети, использующие разную среду передачи данных, например витую пару и радиоволны (беспроводную сеть). Если бы не было модели OSI, то для каждой сети пришлось бы разрабатывать модель взаимодействия, а потом придумывать способ, позволяющий заставить две разные по своей архитектуре сети работать вместе. В случае с моделью OSI не нужно изобретать велосипед. Следует взять за основу уже имеющуюся сеть (в данном случае — Ethernet) и переписать физический уровень. В итоге нам не придется разрабатывать браузеры, почтовые клиенты и другие сетевые приложения для каждой сети — браузеру все равно, какая среда передачи данных используется. Как видите, модель OSI хоть и теоретическая, зато очень полезная.

Рассмотрим ее уровни:

- на *физическом уровне* определяются характеристики электрических сигналов, т. е. описывается физическая среда данных. На этом уровне и происходит физическая передача данных по кабелю или радиоволнам (в зависимости от типа сети). Пример протокола физического уровня: 1000Base-T — спецификация Ethernet, передающая данные по витой паре 5-й категории (о категориях витой пары мы поговорим позднее) со скоростью 1000 Мбит/с;
- *канальный уровень* используется для передачи данных между компьютерами (и другими устройствами, например сетевыми принтерами) одной сети. Пример протокола канального уровня: PPP (Point-to-Point Protocol). Топология сети (шина, звезда и т. д.) определяется как раз на канальном уровне (ранее мы подробно рассмотрели все используемые топологии сетей). На канальном уровне все передаваемые данные разбиваются на части, называемые *кадрами* (frames). Канальный уровень передает кадры физическому уровню, а тот, в свою очередь, отправляет их в сеть.

  На канальном уровне вводится понятие MAC-адреса. *MAC-адрес* — это уникальный аппаратный адрес сетевого устройства (например, сетевого адаптера, точки доступа). Каждому производителю сетевых устройств выделяется свой диапазон MAC-адресов. В мире нет двух сетевых устройств с одинаковыми MAC-адресами;
- *сетевой уровень* используется для объединения нескольких сетей, т. е. для организации межсетевого взаимодействия, — ведь протоколы канального уровня могут работать только в пределах одной сети. Канальный уровень не может передать кадр компьютеру, который находится в другой сети. Понятно, что если бы у нас был только канальный уровень и не было сетевого, мы не смогли бы передавать данные между двумя сетями, например между локальной сетью и Интернетом. Пример протокола сетевого уровня: IP (Internet Protocol). Конечно, IP — это не единственный протокол сетевого уровня, но в этой книге мы будем рассматривать только TCP/IP, поэтому нет смысла упоминать другие протоколы.

  При всем своем желании мы не можем построить огромную сеть, охватывающую весь мир (даже если бы это и удалось, не думаю, что такая сеть работала бы быстро). Поэтому Интернет состоит из совокупности различных сетей, которые объединяются в одно целое маршрутизаторами. Расстояние между сетями исчисляется в количестве переходов пакетов (на сетевом уровне передаваемые данные разбиваются именно на пакеты) через маршрутизаторы. Единица такого перехода называется *хопом* (от англ. *hop*). Количество хопов равно количеству маршрутизаторов между двумя сетями. Например, от моего узла до узла **volia.net** 6 хопов (шесть переходов), что показано на рис. 20.1;
- *транспортный уровень* отвечает за доставку пакетов получателю. Не секрет, что при передаче по каналам связи данные могут быть повреждены или вовсе потеряны. Гарантирует доставку пакета в первозданном виде именно транспортный уровень. На этом уровне осуществляются обнаружение и исправление ошибок, восстановление прерванной связи и некоторые дополнительные сервисы вроде срочной доставки (приоритет пакета) и мультиплексирование нескольких со-

единений. Самым известным и распространенным протоколом транспортного уровня является TCP (Transport Control Protocol);

- *сеансовый уровень* отвечает за установку и за разрыв соединения между компьютерами. На этом уровне также предоставляются средства синхронизации. Сеанс сетевого уровня заключается в установке соединения (при установке стороны, между которыми будут передаваться данные, могут договариваться о дополнительных параметрах связи, например обмениваться ключами), передаче информации и разрыве соединения.

  Многие часто путают сеансы сетевого уровня и сеанс связи. Вы можете установить сеанс связи (например, подключиться к Интернету), но не устанавливать логического соединения, т. е. не запустить браузер для соединения с удаленным узлом;

```
denis@denis-desktop:~$ traceroute volia.net
traceroute to volia.net (82.144.192.47), 30 hops max, 40 byte packets
 1  router.shtorm.net (195.62.14.2)  0.330 ms  0.306 ms  0.295 ms
 2  border.shtorm.com (195.62.14.7)  0.523 ms  0.515 ms  0.504 ms
 3  194.44.13.13 (194.44.13.13)  9.435 ms  9.426 ms  9.415 ms
 4  volia-10G-gw.ix.net.ua (195.35.65.221)  9.618 ms  9.613 ms  9.603 ms
 5  v109.TenGig3-8.diamond.volia.net (82.144.193.192)  9.358 ms  9.562 ms  9.554
 ms
 6  tower.volia.net (82.144.192.47)  9.538 ms  9.251 ms  9.240 ms
denis@denis-desktop:~$
```

Рис. 20.1. Количество переходов от моего узла до узла **volia.net**

- как было отмечено чуть ранее, на сеансовом уровне стороны могут договориться о дополнительных параметрах, были также упомянуты *ключи*. Однако само шифрование и дешифрование данных осуществляется *представительным уровнем*. Пример протокола этого уровня — SSL (Secure Socket Layer);
- последний (самый высокий) уровень — *прикладной*. На этом уровне работает множество разных протоколов, например: HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol) и т. д.

20.2. Что такое протокол

В этой главе довольно часто упоминалось слово "протокол". *Протокол* — это правила, определяющие взаимодействие компьютеров в вычислительной сети. Рассмотрим несколько самых важных протоколов.

В основе Интернета лежит протокол TCP/IP (Transmission Control Protocol/Internet Protocol). Чтобы система смогла работать в Интернете, она должна поддерживать протокол TCP/IP. Вообще говоря, TCP/IP — это совокупность двух протоколов. Протокол TCP, как уже было отмечено ранее, отвечает за корректность передачи данных по Интернету (точнее, по любой сети, использующий этот протокол), т. е. гарантирует доставку данных по сети. Протокол IP используется для адресации компьютеров сети. Дело здесь в том, что у каждого компьютера сети имеется свой

уникальный адрес (IP-адрес), и, чтобы передать данные компьютеру, нужно его IP-адрес знать. Чуть позже мы поговорим о системе доменных имен (DNS, Domain Name System).

Кроме протоколов TCP и IP посетители Интернета работают с теми или иными серверами, использующими следующие протоколы:

- HTTP (Hyper Text Transfer Protocol) — протокол передачи гипертекста. Все Web-серверы Интернета используют протокол HTTP или его безопасную версию — HTTPS (HTTP Secure);
- FTP (File Transfer Protocol) — протокол передачи файлов. Используется для обмена файлами между компьютерами. Вы можете подключиться к FTP-серверу и скачать необходимые вам файлы или же, наоборот, закачать свои файлы на сервер, если вы обладаете надлежащими правами доступа к серверу. В Интернете много публичных FTP-серверов, к которым разрешен анонимный доступ. Как правило, с таких серверов скачивать файлы разрешено всем желающим. Иногда, но очень редко, разрешается и запись файлов на публичный сервер. В состав любой операционной системы входит FTP-клиент — программа ftp. К тому же многие браузеры можно использовать в качестве FTP-клиента;
- SMTP (Simple Mail Transfer Protocol) — простой протокол передачи почты. Используется для отправки почты (e-mail);
- POP (Post Office Protocol) — протокол, используемый для получения сообщений электронной почты;
- IMAP (Internet Message Access Protocol) — еще один протокол для получения почты, но, в отличие от протокола POP, этот протокол позволяет читать почту без ее загрузки на компьютер пользователя. Протокол IMAP, по сути, намного удобнее, чем POP. Ведь почта хранится на сервере, и вы можете получить доступ к ней с любой точки земного шара, используя любой клиент. К тому же IMAP поддерживает поиск писем на сервере, что позволяет найти нужное письмо без загрузки всех писем на свой компьютер. Однако у IMAP есть один существенный недостаток, благодаря которому до сих пор распространен протокол POP — IMAP требует постоянного соединения с сервером. Если нет соединения с сервером, то вы не прочитаете не только новые сообщения, но и те, которые были получены ранее, поскольку все они хранятся на сервере. Так что в случае с IMAP об автономной работе (без подключения к Интернету) можно забыть.

20.3. Адресация компьютеров

Для идентификации узлов Интернета используются IP-адреса. IP-адрес представляет собой четыре числа, разделенные точками (или одно 32-разрядное число, которое записывается в виде четырех восьмиразрядных чисел, разделенных точками, — как кому больше нравится). Нужно сразу отметить, что такая идентификация неоднозначная, поскольку IP-адреса могут быть статическими (постоянными) и динамическими. *Постоянные* (статические) IP-адреса обычно назначаются серверам, а *динамические* — обычным пользователям. Так что сегодня определенный динами-

ческий IP-адрес может быть назначен одному пользователю, а завтра — другому. Поэтому если в случае с аппаратными MAC-адресами еще можно говорить о какой-то однозначности (и то существуют способы подделки MAC-адресов), то IP-адреса по определению однозначными не являются.

Рассмотрим примеры IP-адресов: 127.0.0.1, 192.168.1.79, 111.33.12.99. Как было сказано ранее, IP-адрес — это одно 32-разрядное число или четыре 8-разрядных числа. Возведем 2 в восьмую степень и получим максимальное значение для каждого из четырех восьмиразрядных чисел — 256. Таким образом, учитывая, что некоторые IP-адреса зарезервированы для служебного использования, протокол IP может адресовать примерно 4,3 млрд узлов. Однако с каждым годом количество узлов во Всемирной паутине увеличивается, поэтому была разработана шестая версия протокола IP — IPv6 (если упоминается просто протокол IP, то, как правило, имеется в виду четвертая версия протокола — IPv4). Новый протокол использует 128-битные адреса (вместо 32-битных), что позволяет увеличить число узлов до 10^{12} и количество сетей до 10^9. IPv6-адреса отображаются как восемь групп шестнадцатеричных цифр, разделенных двоеточиями. Вот пример адреса нового поколения: 1628:0d48:12a3:19d7:1f35:5a61:17a0:765d.

> ***Примечание***
>
> Впрочем, массовый переход на IPv6 (который еще называют IPng — IP Next Generation) пока так и не состоялся, хотя его используют несколько сотен сетей по всему миру. В этой книге мы будем рассматривать только протокол IPv4, поскольку, судя по всему, Интернет не перейдет на IPv6 в ближайшие несколько лет. Интересующиеся могут прочитать об IPv6 по адресу: **http://ru.wikipedia.org/wiki/IPv6**.

IP-адреса выделяются *сетевым информационным центром* (NIC, Network Information Center). Чтобы получить набор IP-адресов для своей сети, вам надо обратиться в этот центр. Но, оказывается, это приходится делать далеко не всем. Существуют специальные IP-адреса, зарезервированные для использования в локальных сетях. Ни один узел глобальной сети (Интернета) не может обладать таким "локальным" адресом. Вот пример локального IP-адреса — 192.168.1.1. В своей локальной сети вы можете использовать любые локальные IP-адреса без согласования с кем бы то ни было. Когда же вы надумаете подключить свою локальную сеть к Интернету, вам понадобится всего один "реальный" IP-адрес — он будет использоваться на маршрутизаторе (шлюзе) доступа к Интернету.

Чтобы узлы локальной сети (которым назначены локальные IP-адреса) смогли "общаться" с узлами Интернета, используется специальная технология *трансляции сетевого адреса* (NAT, Network Address Translation). Маршрутизатор получает пакет от локального узла, адресованный интернет-узлу, и преобразует IP-адрес отправителя, заменяя его своим IP-адресом. При получении ответа от интернет-узла маршрутизатор выполняет обратное преобразование, поэтому нашему локальному узлу "кажется", что он общается непосредственно с интернет-узлом. Если бы маршрутизатор отправил пакет как есть, т. е. без преобразования, то его отверг бы любой маршрутизатор Интернета, и пакет так и не был бы доставлен к получателю.

Наверное, вам не терпится узнать, какие IP-адреса можно использовать без согласования с NIC? Об этом говорить пока рано — ведь мы еще ничего не знаем о *клас-*

сах сетей. IP-адреса используются не только для адресации отдельных компьютеров, но и целых сетей. Вот, например, IP-адрес сети — 192.168.1.0. Отличительная черта адреса сети — 0 в последнем октете.

Сети поделены на классы в зависимости от их размеров:

- класс A — огромные сети, которые могут содержать 16 777 216 адресов, IP-адреса сетей лежат в пределах 1.0.0.0–126.0.0.0;
- класс B — средние сети, содержат до 65 536 адресов. Диапазон адресов — от 128.0.0.0 до 191.255.0.0;
- класс C — маленькие сети, каждая сеть содержит до 256 адресов.

Существуют еще и классы D и E, но класс E не используется, а зарезервирован на будущее (хотя будущее — это IPv6), а класс D зарезервирован для служебного использования (широковещательных рассылок).

Представим ситуацию. Вы хотите стать интернет-провайдером. Тогда вам нужно обратиться в NIC для выделения диапазона IP-адресов под вашу сеть. Скажем, вы планируете сеть в 1000 адресов. Понятно, что сети класса C вам будет недостаточно. Поэтому можно или арендовать четыре сети класса C, или одну класса B. Но, с другой стороны, 65 536 адресов для вас — много, и если выделить вам всю сеть класса B, то это приведет к нерациональному использованию адресов. Так что самое время поговорить о *маске сети*. Маска сети определяет, сколько адресов будет использоваться сетью, фактически — маска задает размер сети. Маски полноразмерных сетей классов A, B и C представлены в табл. 20.1.

Таблица 20.1. *Маски сетей классов A, B и C*

Класс сети	Маска сети
A	255.0.0.0
B	255.255.0.0
C	255.255.255.0

Маска 255.255.255.0 вмещает 256 адресов (в последнем октете IP-адреса могут быть цифры от 0 до 255). Например, если адрес сети 192.168.1.0, а маска 255.255.255.0, то в сети могут быть IP-адреса от 192.168.1.0 до 192.158.1.255. Первый адрес (192.168.1.0) называется IP-адресом сети, последний — зарезервирован для широковещательных рассылок. Следовательно, для узлов сети остаются 254 адреса — от 192.168.1.1 до 192.168.1.254.

А вот пример маски сети на 32 адреса: 255.255.255.224
(255 – 224 = 31 + "нулевой" IP-адрес, итого 32).

Предположим, у нас есть IP-адрес произвольной сети, например 192.168.1.0. Как узнать, к какому классу она принадлежит? Для этого нужно преобразовать первый октет адреса в двоичное представление. Число 192 в двоичной системе будет выглядеть так: **110**00000. Проанализируем первые биты первого октета. Если первые биты содержат двоичные цифры 110, то перед нами сеть класса C. Теперь продела-

ем то же самое с сетью 10.0.0.0. Первый октет равен 10, и в двоичной системе он будет выглядеть так: 00001010. Первый бит — 0, поэтому сеть относится к классу A. Опознать класс сети по первым битам первого октета поможет табл. 20.2.

Таблица 20.2. *Опознание класса сети*

Класс сети	Первые биты
A	0
B	10
C	110
D	1110
E	11110

Теперь поговорим о специальных зарезервированных адресах. Адрес 255.255.255.255 является *широковещательным*. Если пакет отправляется по этому адресу, то он будет доставлен всем компьютерам, находящимся с отправителем в одной сети. Можно уточнить сеть, компьютеры которой должны получить широковещательную рассылку, например, таким образом: 192.168.5.255. Этот адрес означает, что пакет получат все компьютеры сети 192.168.5.0.

Вам также следует знать адрес 127.0.0.1. Этот адрес зарезервирован для обозначения локального компьютера и называется *адресом обратной петли*. Если отправить пакет по этому адресу, то его получит ваш же компьютер, т. е. получатель является отправителем, и наоборот. Данный адрес обычно используется для тестирования поддержки сети. Более того, к локальному компьютеру относится любой адрес из сети класса A с адресом 127.0.0.0. Поэтому при реальной настройке сети нельзя использовать IP-адреса, начинающиеся со 127.

А теперь можно рассмотреть IP-адреса сетей, зарезервированные для локального использования. В локальных сетях вы можете использовать следующие адреса сетей:

- 192.168.0.0–192.168.255.0 — сети класса C (всего 256 сетей, маска 255.255.255.0);
- 172.16.0.0–172.31.0.0 — сети класса B (всего 16 сетей, маска 255.255.0.0);
- 10.0.0.0 — сеть класса A (одна сеть, маска 255.0.0.0).

Обычно в небольших домашних и офисных сетях используются IP-адреса из сети класса C, т. е. из диапазона 192.168.0.0–192.168.255.0. Но поскольку назначение адресов контролируется только вами, вы можете назначить в своей локальной сети любые адреса, например адреса из сети 10.0.0.0, даже если у вас в сети всего пять компьютеров. Так что выбор сети — это дело вкуса. Можете себя почувствовать администратором огромной сети и использовать адреса 10.0.0.0.

В следующей главе мы рассмотрим функции для работы с сокетами, а затем напишем наше первое сетевое приложение.

ГЛАВА 21

Программирование сокетов: теория

21.1. Что такое сокет

Сокет — двунаправленный канал обмена данными между двумя компьютерами в сети. Информация по сокету может передаваться в обоих направлениях — от сервера к клиенту и от клиента к серверу.

Все мы знаем, как работать с файлами. Функцией `open()` мы открываем файл. Если файл открыт успешно, то функция возвращает его дескриптор, через который производятся все дальнейшие операции с файлом (чтение, запись).

Сокет очень похож на дескриптор файла, но он является дескриптором соединения, т. е. служит для передачи файлов по сети. Примечательно, что с сетевым соединением в Linux можно работать, почти как с обычным файлом, — с помощью стандартных функций для работы с файлами. Например, вы можете использовать системный вызов `write()` для записи информации в сокет, а вызов `read()` — для чтения информации из сокета. Однако не стоит забывать, что сокет — это не файл в прямом смысле этого слова, поэтому надеяться, что программирование сокетов в Linux — это просто, не стоит.

Прежде чем приступить к написанию реальных приложений, нужно разобраться с общим алгоритмом программирования сокетов:

1. Подготовка сокета функцией `socket()`.
2. Связывание сокета с портом функцией `bind()`.
3. Установка связи с удаленным компьютером (на стороне сервера используется функция `accept()`, на стороне клиента — функция `connect()`).
4. Обмен данными.
5. Завершение сеанса связи — функции `shutdown()` и `close()`.

Библиотечные функции, необходимые для работы с сокетами, описаны в заголовочном файле socket.h:

```
#include <sys/socket.h>
```

21.2. Создание и связывание сокета

Рассмотрим первые два пункта нашего алгоритма. Первым делом рассмотрим функцию `socket()`, определенную в socket.h. Также желательно подключить sys/types.h:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

> ***Примечание***
>
> Стандарт POSIX.1.2001 не требует подключение файла sys/types.h и этот заголовочный файл можно не подключать при программировании в Linux. Однако, если вы заботитесь о переносимости вашего приложения и планируете портировать его в другую UNIX-подобную систему (например, в BSD), подключите sys/types.h.

Функция `socket()` создает сокет. Первый параметр определяет набор протоколов. Если особо не вдаваться в подробности, то вам нужно всегда в качестве этого параметра передавать значение `AF_INET`, что означает использование стека протоколов TCP/IP.

Второй параметр устанавливает режим работы сокета:

- `SOCK_STREAM` — режим с установкой соединения. Подходит для TCP/IP;
- `SOCK_DGRAM` — режим без установки соединения. Подходит для UDP;
- `SOCK_RAW` — если вы знаете, что делаете (данный режим работы сокета, как вы уже догадались, в этой книге рассматриваться не будет).

Третий параметр задает протокол, обычно его можно установить в 0, чтобы был выбран протокол по умолчанию в зависимости от режима работы сокета. Если вы выбрали `SOCK_STREAM`, то будет использоваться протокол TCP, а если вы выбрали `SOCK_DGRAM`, то будет выбран протокол UDP.

> ***Примечание***
>
> Хотя я и обещал, что `SOCK_RAW` рассматриваться не будет, напишу о нем пару строк. Для использования этого режима сокета нужно подключить заголовочный файл netinet/in.h. Номер протокола при режиме `SOCK_RAW` для функции `socket()` можно взять из файла /etc/protocols.

Функция `socket()` возвращает дескриптор сокета — целое положительное число или же –1, если сокет создать не удалось.

После создания сокета его нужно связать с локальным портом, для этого используется функция `bind()`:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Первый параметр — это дескриптор сокета, второй — указатель на структуру типа `sockaddr`, содержащую адрес компьютера, номер порта и другую необходимую для установки соединения информацию, третий — размер структуры `sockaddr` в байтах.

Все структуры типа `sockaddr` определены в файле socket.h (/usr/incude/sys/socket.h):

```
# define __SOCKADDR_ALLTYPES \
  __SOCKADDR_ONETYPE (sockaddr) \
  __SOCKADDR_ONETYPE (sockaddr_at) \
  __SOCKADDR_ONETYPE (sockaddr_ax25) \
  __SOCKADDR_ONETYPE (sockaddr_dl) \
  __SOCKADDR_ONETYPE (sockaddr_eon) \
  __SOCKADDR_ONETYPE (sockaddr_in) \
  __SOCKADDR_ONETYPE (sockaddr_in6) \
  __SOCKADDR_ONETYPE (sockaddr_inarp) \
  __SOCKADDR_ONETYPE (sockaddr_ipx) \
  __SOCKADDR_ONETYPE (sockaddr_iso) \
  __SOCKADDR_ONETYPE (sockaddr_ns) \
  __SOCKADDR_ONETYPE (sockaddr_un) \
  __SOCKADDR_ONETYPE (sockaddr_x25)
```

Нас интересует структура `sockadd_in`, подходящая для TCP/IPv4, для TCP/IPv6 следует использовать структуру типа `sockadd_in6`. Интересующие нас структуры (описаны в netinet/in.h) выглядят так:

```
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;             /* Номер порта */
    struct in_addr sin_addr; /* IP-адрес  */

    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
  };

/* Для IPv6.  */
struct sockaddr_in6
  {
    __SOCKADDR_COMMON (sin6_);
    in_port_t sin6_port;            /* Порт транспортного уровня */
    uint32_t sin6_flowinfo;         /* Информация  потока IPv6 */
    struct in6_addr sin6_addr;              /* Адрес IPv6 */
    uint32_t sin6_scope_id;         /* IPv6-идентификатор */
  };
```

Изучением структуры для IPv6 при желании займетесь сами — не вижу смысла рассматривать IPv6, поскольку он еще реально не используется. Пространства адресов IPv4 оказалось достаточно на данный момент.

Разберемся с полями структуры `sockaddr_in`:

- `sin_` — набор используемых протоколов, для TCP/IP это поле должно содержать значение `AF_INET`;

- `sin_port` — номер порта;
- `sin_addr` — структура, содержащая IP-адрес;
- `sin_zero` — обычно не используется.

Структура `struct in_addr`, определяющая адрес узла, также описана в файле netinet/in.h:

```
struct in_addr
  {
    in_addr_t s_addr;
  };
```

Обычно поле `s_addr` должно содержать значение `INADDR_ANY`. В этом случае функция `bind()` автоматически привяжет адрес локального компьютера к сокету и нам вообще не нужно указывать его явно. Так ваша программа будет более универсальной. Из соображений гибкости конфигурации можете предусмотреть конфигурационный файл, содержащий значение для этого поля: явная установка IP-адреса понадобится, если в системе несколько сетевых интерфейсов и вы хотите точно указать, к какому из них нужно привязать сокет.

Функция `bind()` возвращает 0 в случае успеха и –1, если произошла ошибка. Рассмотрим небольшой пример:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
...
int sock;

sock = socket (AF_INET, SOCK_STREAM, 0);
if (sock==-1) {
printf("Ошибка создания сокета\n");
exit(1);
}

struct sockaddr_in client;
...
client.sin_family = AF_INET;
client.sin_addr.s_addr = INADDR_ANY;
client.sin_port = 1235;

bind (sock, (struct sockaddr *)&client, sizeof(client));
```

21.3. Установление связи с удаленным компьютером

После того как сокет создан и связан с портом, нужно установить связь с удаленным компьютером. На стороне клиента используется только функция `connect()`, а

на стороне сервера могут использоваться `listen()` и `accept()`. Первая ждет клиента, а вторая — подтверждает запрос клиента на установку соединения.

Начнем с функции `listen()`:

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Первый параметр — дескриптор сокета, а второй — максимальное количество запросов на установку связи (максимальное количество клиентов). В случае успеха возвращается 0.

Если максимальное число клиентов не превышено, сервер может принять запрос клиента с помощью функции `accept()`. Данная функция используется только при работе в режиме с установлением соединения:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Первый параметр — это дескриптор сокета, второй — это указатель на структуру, в которой можно разместить адрес клиента. Структуру до вызова `accept()` инициализировать не нужно. Последний параметр — размер структуры, указанной во втором параметре.

Функция `accept()` извлекает из очереди `listen()` запрос на соединение и создает новый сокет, который будет использоваться для обмена данными с клиентов:

```
// Получаем сокет клиента
sock2 = accept (sock1, (struct sockaddr *)&client, &ans_len);
// Передаем клиенту информацию
write (sock2, MSG_TO_SEND, sizeof(MSG_TO_SEND));
```

В случае ошибки `accept()` возвращает отрицательное значение. В случае успеха возвращается 0, а структура, указанная во втором параметре, будет содержать IP-адрес клиента.

Теперь рассмотрим функцию `connect()`, которая используется клиентов для отправки запроса на подключение к серверу:

```
int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

Как обычно, первый параметр — это дескриптор сокета, второй — структура с адресом сервера (и номером порта сервера), а третий — длина структуры, указанной во втором параметре.

Как обычно, в случае успеха возвращается 0, в случае ошибки — –1. Пример вызова `connect`:

```
struct sockaddr_in server;
struct hostent *h;
...
/* Определяем IP-адрес сервера */
h = gethostbyname ("server.ru");
```

```
memcpy ((char *)&server.sin_addr,h->h_addr,h->h_length);
/* Порт сервера */
server.sin_port = 3333;
/* Семейство протоколов */
server.sin_family = AF_INET;

/* Подключаемся */
connect (sock, (struct sockaddr *)&server, sizeof(server));
```

Обратите внимание: чтобы не указывать IP-адрес сервера, мы его вычислили с помощью функции `gethostbyname()`, передав ей символьное имя сервера. Узнать свой собственный адрес можно с помощью функции `getsockname()`:

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

Этой функции нужно передать три параметра — дескриптор сокета, адрес структуры, которая будет содержать данные о нашем узле и длину структуры, указанную во втором параметре.

21.4. Передача данных

Мы установили подключение с удаленным компьютером. Теперь нужно передать данные. Это можно сделать обычными функциями `read()` и `write()`, только вместо дескриптора файла нужно указать дескриптор сокета. Однако лучше использовать функции `send()` и `recv()`, предназначенные именно для работы с сокетами.

Если у вас сокет работает в режиме без установки соединения (протокол UDP), тогда вам нужно использовать функции `sendto()` и `recvfrom()`. Первая функция отправляет данные, а вторая — принимает. Функция `sendto()` вместе с данными позволяет указать адрес получателя, а `recvfrom()` возвращает не только полученные данные, но и адрес отправителя.

Сначала мы рассмотрим функции `send()` и `recv()`:

```
ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Первая функция используется для отправки данных, вторая — для чтения. Первый параметр обеих функций — это дескриптор сокета. Второй — указатель на буфер данных. В случае с функцией `send()` данный параметр является константой, а вот в случае с `recv()` в `buf` будут записаны принимаемые данные. Третий параметр — это размер буфера, а последний — флаги.

Флаги различны для функций отправки/получения данных. Назначение флагов вы можете прочитать в справочной системе:

```
man 2 send
man recv
```

Обе функции возвращают количество отправленных/принятых байтов или –1 в случае ошибки.

У обеих функций есть одна особенность: если функция `send()` не может отправить данные (например, по причине переполнения буфера-приемника), то она переводит программу в режим ожидания, пока буфер не будет готов принять данные. Функция `recv()` работает аналогично: если она ничего не может прочитать, она будет ждать до тех пор, пока в сокете не появятся данные. Чтобы исправить такое поведение функций, нужно указать флаг `MSG_DONTWAIT`. С назначением остальных флагов вы ознакомитесь в справочной системе.

Функции `sendto()` и `recvfrom()` работают в режиме без установки соединения:

```
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

Функции аналогичны функциям `send()` и `recv()`, но небольшие изменения все-таки есть. Поскольку функции работают в режиме без установки соединения, то им нужно передать структуру типа `sockaddr`, содержащую адрес компьютера, которому нужно отправить данные (для `sendto`) или от которого нужно получить данные (для `recvfrom`). Последние параметры обеих функций задают размер структуры типа `sockaddr`.

21.5. Завершение сеанса связи

Завершить сеанс связи можно системным вызовом `close()`, который закрывает сокет, как обычный файл, но такое завершение сеанса считается довольно грубым. Рекомендуется сначала вызвать функцию `shutdown()`, а затем уже вызывать `close()`:

```
int shutdown(int s, int how);
```

Функция `shutdown()` определяет, как будет завершен сеанс связи. Первый параметр — это, собственно, сам сокет (точнее, его дескриптор), а второй — метод завершения соединения:

- `SHUT_RD` — передать данные, отправка которых уже началась, но которые еще не успели быть переданы на удаленный компьютер. Больше данные приниматься не будут;
- `SHUT_WR` — передать данные и запретить прием данных через сокет;
- `SHUT_RDWR` — передать данные и запретить обмен через сокет: ни приема, ни передачи.

Функция, как обычно, возвращает 0 в случае успеха и –1 в случае ошибки. В следующей главе мы рассмотрим создание сетевого приложения.

ГЛАВА 22

Программирование сокетов: практика

22.1. Создание приложения клиент/сервер

22.1.1. Программа-сервер

Настало время задействовать на практике полученные знания. Мы напишем простую программу-сервер, ожидающую подключения клиента. После того как клиент подключится, сервер отправит ему приветствие и будет ждать от клиента любой текстовой строки. Сервер прочитает данные, полученные от клиента, и выведет их на экран.

Нужно отметить, что программы (клиент и сервер) будут сугубо демонстрационными и их исходный код, исходя из соображений демонстрации обмена данными между клиентом и сервером, будет существенно упрощен. В частности мы не будем производить обработку ошибок. Если программа-клиент запустится и сразу завершит работу, проверьте, правильно ли вы указали адрес сервера и номер порта (если вы будете использовать номер порта и имя сервера, отличные от используемых в листингах программ).

Исходный код программы-сервера представлен в листинге 22.1. Каждая строка в листинге 22.1 прокомментирована, поэтому внимательно читайте комментарии, чтобы понять, что делает программа.

Листинг 22.1. Программа server.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
```

```
/* Порт сервера, к этому порту должен подключаться клиент */
#define PORT 3333
/* Размер буфера */
#define BUF_SIZE 64
/* Приглашение сервера, его увидит клиент после подключения к серверу */
#define MSG_TO_SEND "My Simple Server v0.0.1\n"

int main () {
/* Сокеты для клиента и сервера */
int server_socket, client_socket;

/* Длина ответа и счетчик клиентов */
int answer_len, count=0;

/* Буфер для чтения данных */
char buffer[BUF_SIZE];
/* Структуры sockaddr_in для клиента и сервера (sin) */
struct sockaddr_in sin, client;

/* Создаем сокет сервера */
server_socket = socket (AF_INET, SOCK_STREAM, 0);
/* Заполняем память, см. man memset
   данная операция используется для очистки структуры */
memset ((char *)&sin, '\0', sizeof(sin));

/* Заполняем структуру sin */
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = PORT;

/* Связываем сокет сервера с портом 3333 */
bind (server_socket, (struct sockaddr *)&sin, sizeof(sin));

/* Сообщаем о своем запуске */
printf("Server started.\n");

/* Слушаем сокет сервера */
listen (server_socket, 3);

/* Запускаем бесконечный цикл для принятия запросов от клиентов,
завершить работу программы можно, нажав <Ctrl>+<C> */
while (1) {
       /* Вычисляем длину ответа */
       answer_len = sizeof(client);
       /* Принимаем запрос клиента */
       client_socket = accept (server_socket,
                      (struct sockaddr *) &client, &answer_len);
```

```
    /* Отправляем клиенту приветствие */
      write (client_socket, MSG_TO_SEND, sizeof(MSG_TO_SEND));
      /* Увеличиваем счетчик клиента */
      count++;
      /* Читаем ответ клиента в буфер,
      переменная anser_len будет содержать к-во прочитанных байтов */
      answer_len = read (client_socket, buffer, BUF_SIZE);
      /* Выводим на стандартный вывод ответ клиента */
      write (1, buffer, answer_len);
      /* Выводим порядковый номер клиента */
      printf("Client no %d\n",count);
      /* Разрываем соединение */
      shutdown (client_socket, 0);
      /* Закрываем сокет */
      close (client_socket);
 };

return 0;
}
```

Как видите, программа-сервер получилась компактной и не очень сложной для восприятия. Программа выводит только строку, полученную от клиента, и порядковый номер клиента. При желании вы можете усложнить программу, добавив вывод адреса клиента:

```
fprintf(stdout, "Client connected: %s\n", inet_ntoa(client.sin_addr));
```

Можно добавить обработку ошибок при создании сокета, его привязке и вызове `accept()`:

```
if (server_socket = socket (AF_INET, SOCK_STREAM, 0) < 0) {
       printf("Failed to create socket");
       exit(1);
}
...
if (bind (server_socket, (struct sockaddr *)&sin, sizeof(sin)) <0) {
       printf("Failed to bind the server socket");
       exit(1);
}
...
/* Строку нужно добавить после вызова accept */
if (client_socket < 0)
{
       printf("Failed to accept client connection");
       exit(1);
}
```

Одним словом, усовершенствование программы остается на ваше усмотрение, а мы тем временем напишем программу-клиент.

22.1.2. Программа-клиент

Программа-клиент представлена в листинге 22.2. Программа подключается к серверу, выводит полученное от сервера приветствие на экран, затем программа передает строку, указанную в константе `MSG`, серверу и завершает работу. Сначала я хотел читать строку со стандартного ввода, но потом стало лень при отладке программы что-то вводить, поэтому демонстрационная программа ограничивается передачей одной и той же строки. Если есть огромное желание, можете добавить вызов `scanf()` и читать строку со стандартного ввода.

Листинг 22.2. Программа client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>

/* Имя сервера (обычное доменное имя, не IP-адрес),
номер порта сервера, номер порта клиента — они должны отличаться */
#define SERVER_HOST "localhost"
#define SERVER_PORT 3333
#define CLIENT_PORT 3334
/* Строка, которая будет передана серверу */
#define MSG "Denis\n"

main () {
int sock;                           /* Сокет */
int answer_len;              /* Длина ответа сервера */
int BUF_SIZE = 64;           /* Размер буфера */

char buffer[BUF_SIZE];       /* Буфер */
/* Структура для разрешения доменного имени в IP-адрес */
struct hostent *h;
/* Структуры адреса для клиента и сервера */
struct sockaddr_in client, server;

/* Создаем сокет */
sock = socket (AF_INET, SOCK_STREAM, 0);
/* Очищаем структуру client */
memset ((char *)&client, '\0', sizeof(client));

/* Заполняем структуру */
client.sin_family = AF_INET;
```

```
client.sin_addr.s_addr = INADDR_ANY;
client.sin_port = CLIENT_PORT;

/* Связываем сокет клиента с портом клиента */
bind (sock, (struct sockaddr *)&client, sizeof(client));
memset ((char *)&client, '\0', sizeof(server));

/* Получаем IP-адрес сервера */
h = gethostbyname (SERVER_HOST);
/* Заполняем структуру адреса для сервера */
server.sin_family = AF_INET;
/* Устанавливаем адрес сервера */
memcpy ((char *)&server.sin_addr,h->h_addr,h->h_length);
/* Устанавливаем порт сервера */
server.sin_port = SERVER_PORT;

/* Подключаемся к серверу */
connect (sock, (struct sockaddr *) &server, sizeof(server));

/* Получаем приветствие сервера */
answer_len = recv (sock, buffer, BUF_SIZE, 0);
/* Выводим приветствие на стандартный вывод */
write (1, buffer, answer_len);
/* Отправляем серверу данные */
send (sock, MSG, sizeof(MSG), 0);

/* Закрываем сокет и завершаем работу программы */
shutdown (sock, 0);
close (sock);
exit (0);
}
```

На рис. 22.1 наши программы изображены в действии. На нижнем терминале запущена программа-сервер. Как видите, программа сообщила о своем запуске, далее приступила к обработке клиентов. Всего к серверу подключалось два клиента, каждый из которых передавал одну и ту же строку — "Denis" (так и было задумано). В верхнем правом углу изображен терминал, на котором производилась сборка обеих программ и запуск двух клиентов. Клиент получил от сервера приглашение, вывел его на консоль, а затем передал серверу строку. Сервер получил строку от клиента и тоже вывел ее на консоль.

22.2. Параметры сокета

Сами понимаете, написанием двух простых программ мы не ограничимся. Для написания полноценных приложений клиент/сервер в Linux нужно немного углубиться. Начнем с параметров сокета.

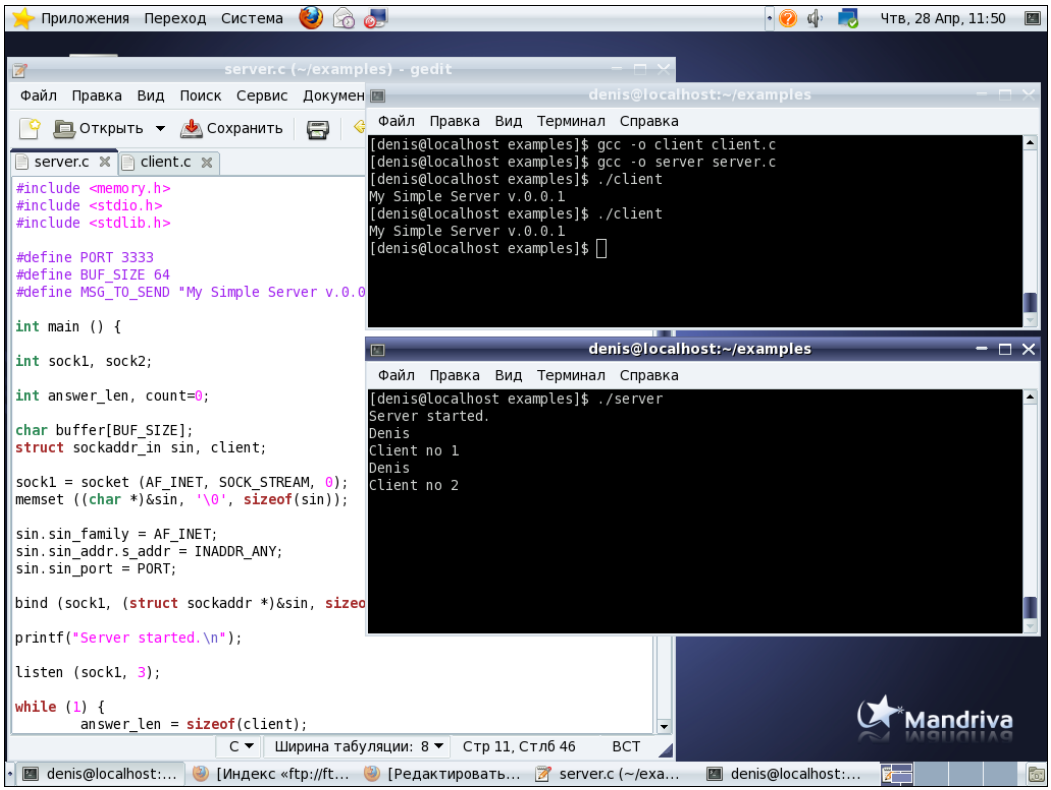


Рис. 22.1. Программы в действии

Для работы с опциями сокетами используются две функции:

- `getsockopt()` — получение опций сокета;
- `setsockopt()` — установка опций сокета.

Прототипы этих двух функций выглядят так:

```
int getsockopt (int sd, int level,
      int option_name,
      void *restrict option_value,
      socklen_t *restrict option_len);

int setsockopt (int sd, int level,
      int option_name,
      const void *option_value,
      socklen_t option_len);
```

Первый параметр — это дескриптор сокета. Второй параметр — это уровень сокета, вы всегда будете использовать уровень `SOL_SOCKET` — другого не дано. Далее следует имя опции (имя вы будете указывать в виде константы типа `int`), которую вы хотите прочитать (функция `get*`) или установить (функция `set*`). Функция `getsockopt()` читает опцию, заданную третьим параметром, в переменную, задан-

ную четвертым параметром (option_value), длина этой переменной будет записана в переменную option_len. Функция setsockopt() устанавливает значение опции, указанной в третьем параметре, новое значение опции указывается параметром option_value, последний параметр, как и в случае с getsockopt(), означает длину опции.

Набор опций зависит от самого сокета. Поскольку мы используем TCP/IP, мы можем устанавливать опции, представленные в табл. 22.1.

Таблица 22.1. *Параметры сокета*

Название опции	Описание
SO_DEBUG	Включает (1) / выключает (0) запись отладочной информации для сокета
SO_BROADCAST	Включает (1) / выключает (0) отправку широковещательных сообщений
SO_REUSEADDR	Параметр разрешает (1) / запрещает (0) использование локальных адресов
SO_KEEPALIVE	Если SO_KEEPALIVE = 1, протокол TCP будет периодически передавать сообщение "keepalive probe" на соединенный сокет. Если адресат будет не в состоянии ответить на это сообщение, соединение считается разорванным
SO_SNDBUF	Параметр устанавливает размер буфера отправки, значение целого типа
SO_RCVBUF	Параметр устанавливает размер буфера приема, значение целого типа
SO_SNDTIMEO	Позволяет установить тайм-аут для отправки сообщений. По умолчанию тайм-аут равен 0. Нужно передать значение типа struct timeval
SO_RCVTIMEO	Позволяет установить тайм-аут для приема сообщений. По умолчанию тайм-аут равен 0, т. е. его вообще нет. Нужно передать значение типа struct timeval
TCP_NODELAY	Используется для отключения (значение 1) механизма буферизации сообщений, т. е. они будут отправляться без задержки. Для включения механизма буферизации нужно указать значение 0
TCP_MAXSEG	Используется для установки максимального сегмента данных. Значение целого типа
TCP_NOPUSH	Не использовать проталкивание (1)
TCP_NOOPT	Не использовать опции TCP (1). Для использования опций передайте значение 0

Функция setsockopt() возвращает 0 в случае успешной установки параметра функции, в случае ошибки возвращается –1, а переменная errno устанавливается так:

- EBADF — неверный дескриптор сокета;
- ENOTSOCK — указанный дескриптор является дескриптором файла, а не сокета;
- EFAULT — нет доступа к адресу, на который указывает указатель optval.

Функция getsockopt() возвращает значение параметра. Кроме вышеперечисленных параметров функция getsockopt() может использовать следующие параметры:

- SO_ERROR — возвращает номер ошибки (номер ошибки будет в возвращаемом значении);
- SO_TYPE — возвращает тип сокета.

Рассмотрим пример чтения и установки опций сокета:

```
int sock;                    /* Дескриптор сокета */
int optval;                  /* Значение опции */
int optlen;                  /* Длина optval  */

int new_buffsize = 8192;     /* Новый размер буфера отправки */

/* Создаем сокет  */
sock = socket(AF_INET, SOCK_STREAM, 0);

/* Читаем длину буфера отправки */
optlen = sizeof(optval);
getsockopt(sock, SOL_SOCKET, SO_SNDBUF, &optval, &optlen);

printf("Old value of SO_SNDBUF: %d\n", optval);

/* Изменяем длину буфера отправки */
setsockopt(sock, SOL_SOCKET, SO_SNDBUF,
           &new_buffsize, sizeof(new_buffsize));

/* Выводим измененную информацию */
getsockopt(sock, SOL_SOCKET, SO_SNDBUF, &optval, &optlen);
printf("New value of SO_SNDBUF %d\n", optval);
```

22.3. Сигналы, связанные с сокетами

Полноценное серверное приложение невозможно написать без обработки сигналов операционной системы. Кроме обычных сигналов, уже рассмотренных в *главе 10*, сервер может реагировать на сигналы, связанные с сокетами. С сокетами связаны сигналы SIGIO, SIGURG, SIGPIPE:

- SIGIO — сокет готов к вводу/выводу, сигнал посылается процессу, который связан с сокетом;
- SIGURG — сигнал посылается, когда сокет получил экспресс-данные (в книге экспресс-данные не рассматриваются);
- SIGPIPE — посылается, если запись в сокет больше невозможна.

Пример обработки сигнала:

```
#include <signal.h>
...
/* Будем обрабатывать сигнал SIGPIPE */

/* Обработчик сигнала */
sig_handler()
```

```
{
  printf("Получен SIGPIPE \n");
}

main()
{
  int sock;          /* Дескриптор сокета */

/* Устанавливаем обработчик сигнала SIGPIPE */
  signal(SIGPIPE, sig_handler);

/* Работаем с сокетом */

}
```

22.4. Неблокирующие операции

Некоторые функции для работы с сокетами блокируют программу, если удаленный процесс не осуществил требуемую операцию. Программа может блокироваться функциями `accept()`, `connect()`, `read()`, `write()`. Иногда блокирование операций нежелательно, поскольку за время блокировки можно было бы выполнить полезные действия, например обработать информацию, полученную ранее.

Сокет можно перевести в неблокирующий режим с помощью системного вызова `ioctl()`:

```
ioctl(socket, FIONBIO, 1);
```

После такого вызова функции `accept()`, `connect()` и функции чтения/записи данных (`read()`, `write()`, `recv()`, `recvfrom()` и др.) работают особенно. Функция `accept()` сразу завершает свою работу с ошибкой `EWOULDBLOCK`, функция `connect()` тоже завершает работу, но с другой ошибкой — `EINPROGRESS`.

Что же касается функций чтения, то их нужно вызывать периодически, чтобы не потерять данные в неблокирующем режиме: ведь если функция `read()` завершилась потому, что не было данных, это не означает, что через секунду данные не поступят — их нужно будет прочитать. Поэтому не забывайте периодически вызывать функции чтения.

ЧАСТЬ VI

Создание графического интерфейса средствами TCL/Tk

Глава 23.	Введение в TCL/Tk
Глава 24.	Синтаксис TCL
Глава 25.	Работа с файлами
Глава 26.	Понятие о виджетах
Глава 27.	Основные элементы графического интерфейса
Глава 28.	Многооконный интерфейс
Глава 29.	Практический пример

Шестая часть книги посвящена языку программирования TCL и библиотеке Tk, которая используется вместе с TCL для создания графического интерфейса пользователя. Далеко не всегда нужно создавать программы C, для создания простых утилит вполне хватит возможностей TCL/Tk.

ГЛАВА 23

Введение в TCL/Tk

23.1. Знакомство с TCL

TCL (Tool Command Language) — скриптовый язык высокого уровня. Язык TCL очень часто применяется с графической библиотекой TK (Took Kit) для создания графического интерфейса.

Вспомните, с чего начиналась наша книга. Правильно. Со скриптовых языков командной оболочки. Затем мы рассмотрели пакет dialog для создания псевдографического интерфейса пользователя. Однако создать с dialog полноценный графический интерфейс невозможно. Сейчас мы рассмотрим полноценный скриптовый язык TCL, возможности которого превышают возможности языка командной оболочки и библиотеку Tk, которую мы будем использовать для создания графического интерфейса. В следующей части мы рассмотрим графическую библиотеку GTK, которую используют для создания графического интерфейса при программировании на C. Что лучше? Конечно же, намного правильнее использовать связку C + GTK, потому что в этом случае программа будет скомпилирована, а это означает, что она будет выполняться быстрее. Однако есть множество областей, где возможностей TCL/Tk будет вполне достаточно и вам не нужно использовать более мощные средства для этого. Примеры программ, которые целесообразно разрабатывать с использованием TCL/Tk: создание оболочек для консольных программ, конфигураторы системы, различные информационные программы, вспомогательные системные утилиты. Более того, TCL, как встраиваемый язык, нашел применение в сфере САПР (CAD/CAM), также он используется как средство настройки баз данных и является языком автоматизации во всех ведущих пакетах разработки микросхем. Одним словом, знать TCL — полезно. А библиотека Tk — приятный бонус, позволяющий разработать полноценный графический интерфейс для ваших TCL-программ.

23.2. Установка TCL/Tk

Пара TCL/Tk входит в состав всех дистрибутивов Linux (исключение могут составить экзотические и узкоспециализированные дистрибутивы) и в большинстве слу-

чаев устанавливается по умолчанию. Это означает, что вам вообще ничего не придется делать для установки TCL/Tk.

На всякий случай просмотрите список установленных пакетов и убедитесь, что пакеты tcl и tk установлены. Если это не так, установите их. Например, в Ubuntu для установки этих пакетов можно использовать команду:

```
sudo apt-get install tcl tk
```

В других дистрибутивах воспользуйтесь вашим менеджером пакетов (`yum`, `zypper` и т. д.).

23.3. Первая программа

Синтаксис TCL будет подробно описан в следующей главе, а пока напишем небольшую программу, чтобы вы убедились, что программирование на TCL — довольно простая задача.

Первым делом выясним, в каком каталоге находится интерпретатор `tclsh`:

```
which tclsh
```

Обычно `tclsh` находится в каталоге /usr/bin. В этом же каталоге будет программа `wish`, которая используется для выполнения графических TCL-программ. Это и есть интерпретатор Tk.

Сейчас мы напишем простенькую программу, выводящую графическое окно с кнопкой **Exit**. Это намного лучше, чем разрабатывать программу в стиле "Hello, world!", состоящую из всего одной инструкции. А так мы создадим пусть и простую, но графическую программу. Программа простая, т. к. на данном этапе, когда вы еще не знакомы с синтаксисом TCL, вряд ли вам будет понятна сложная программа. Итак, в листинге 23.1 представлено ваше первое TCL/Tk-приложение.

Листинг 23.1. Ваше первое TCL/Tk-приложение

```
#!/usr/bin/wish
label .l -width [string length "My First Tk-application"] -text "My First Tk-
application";
button .b -text "Exit" -command exit;
pack .l -padx 40 -pady 10
pack .b -padx 40 -pady 10
```

Первая строка — вызов интерпретатора Tk. Вторая — формирование текстовой надписи "My First Tk-application". В квадратных скобках происходит вычисление длины этой строки. Фактически всю эту конструкцию можно было бы заменить обычным числом, и тогда бы вторая строка программы выглядела так:

```
label .l -width 23 -text "My First Tk-application";
```

Если вы будете использовать расширенный вариант второй строки, то вы поместите все символы команды в одну строку, не нужно ничего переносить. Понятно, что длинные строки не всегда будут помещаться на странице книги.

Третья строка формирует кнопку с надписью **Exit**, при нажатии этой кнопки происходит выполнение команды (задается параметром `-command`) `exit`.

Далее командами `pack` мы задаем расположение элементов (надписи и кнопки) относительно окна. Подробно параметры `-padx` и `-pady`, а также другие стандартные параметры графических виджетов, будут рассмотрены в *главе 27*.

Сохраните файл как first.tcl и введите команды:

```
chmod +x first.tcl
./first.tcl
```

Первая команда делает файл first.tcl исполнимым, а вторая — запускает его. В результате вы увидите окно, изображенное на рис. 23.1.

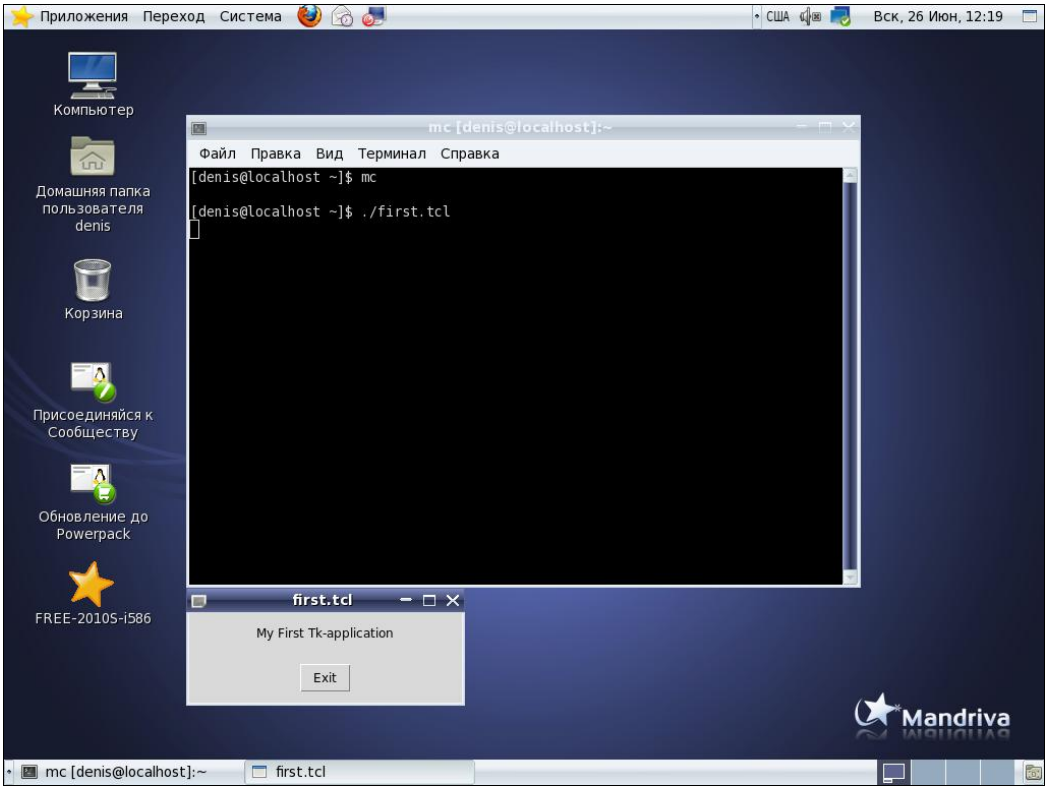


Рис. 23.1. Созданная нами программа

В следующей главе мы рассмотрим синтаксис языка TCL, а совершенствовать наши познания в области графического интерфейса продолжим только в *главе 26*, поскольку нет смысла рассматривать графические виджеты, если сам синтаксис TCL еще не изучен.

Для запуска наших TCL-сценариев без графического интерфейса мы будем использовать интерпретатор `tclsh`:

```
tclsh <имя файла TCL-программы>
```

Начинающим программистам, еще не освоившим синтаксис TCL, рекомендуется поначалу не создавать TCL-сценарии, а запускать `tclsh` в интерактивном режиме:

```
tclsh
```

После этого вы увидите приглашение для ввода TCL-команд:

```
%
```

В интерактивном режиме вы видите ошибку сразу же после нажатия <Enter>, что довольно удобно. А если ошибок нет, то вы сразу видите результат операции без необходимости выводить значение переменной с помощью команды `puts` (будет рассмотрена в следующей главе — это аналог функции `printf()` в C).

ГЛАВА 24

Синтаксис TCL

24.1. Знакомство с синтаксисом TCL

24.1.1. Формат TCL-сценария

Синтаксис TCL-программы больше похож на синтаксис командной оболочки, чем на синтаксис классического языка программирования. Хорошо это или плохо — сложный вопрос. Синтаксисы языков C, Java и PHP немного похожи. Представьте, что у всех языков программирования один синтаксис, но разные наборы библиотек: один язык является системным, другой — Web-ориентированным и т. д. В таком случае существенно упростилась бы подготовка программистов, но было бы очень скучно — ничего нового, ничего интересного, все однообразное. А когда синтаксис у всех языков программирования разный, бывает очень интересно на досуге познакомиться с каким-то языком программирования.

TCL-программа представляет собой список команд, разделенных переводом строки (`\n`) или точкой с запятой. Каждая команда имеет формат:

```
имя_команды аргумент1 ... аргументN
```

Посмотрите на листинг нашей первой программы (листинг 23.1) — он действительно является списком команд с параметрами. Единственное, что отличает программу first.tcl от простого списка команд, — это наличие квадратных скобок, в которых производится вычисление выражения.

Если писать сценарий так, чтобы в каждой строке была только одна команда, то первое слово каждой строки будет именем команды, все остальные — аргументами команды. Любой аргумент команды может быть заменен другой командой, помещенной в квадратные скобки. В случае со сценарием first.tcl в квадратных скобках находится команда, вычисляющая длину строки. Любые аргументы в фигурных скобках передаются команде как есть, в виде одного агумента.

Комментарии начинаются с символа решетки `#`. Комментарий может быть как в начале строки, так и в ее конце, например:

```
# Выводим строку
pust "Hello, world!"; # Привет, мир!
```

В TCL-сценариях символы, представленные в табл. 24.1, имеют особое значение.

Таблица 24.1. *Специальные символы*

Символ	Значение
`$`	Подстановка значения переменной
`[]`	Подстановка результата выполнения команды, указанной внутри скобок
`""`	Группирует аргументы в один с подстановкой значений переменных
`{}`	Группирует аргументы в один без подстановки значений переменных
`\`	Экранирует следующий символ или вызывает подстановку управляющего символа

24.1.2. Команды *puts* и *format*: вывод и форматирование строки

Прежде чем продолжить знакомство с синтаксисом языка TCL, нам нужно познакомиться с командой `puts`, которая используется для вывода строк на консоль, и которую мы будем применять для демонстрации наших примеров. По сути, команда `puts` является аналогом `printf()` в C. Команда `puts` только выводит строку, а отформатировать вывод можно с помощью команды `format`.

Команда `puts` обеспечивает запись в поток, по умолчанию (если поток явно не указан) используется поток stdout:

```
puts "Hello, world!";
```

Вывод на stdout явно:

```
puts stdout "Hello!";
```

С помощью параметра `-nonewline` мы можем запретить вывод символа новой строки (`/n`) после вывода нашей строки:

```
puts -nonewline stdout "Hello!";
```

С помощью `puts` можно легко добавить информацию в файл. Однако работа с файлами будет рассмотрена в *главе 25*, а сейчас рассмотрим пример добавления информации в журнал my.log:

```
set chan [open my.log a]
set timestamp [clock format [clock seconds]]
puts $chan "$timestamp — Message"
close $chan
```

Первая команда присваивает переменной `chan` файловый дескриптор открытого в режиме `a` (от *append*) файла my.log, затем переменной `timestamp` присваивается текущее время, после чего мы выводим в файл (с помощью `puts`) строку вида "время — Message". Последняя команда закрывает файл.

Команда `format` форматирует строку в стиле `sprintf()` языка C. Синтаксис вызова `format` следующий:

```
format строка_формата аргумент1 ... аргументN
```

Строка формата аналогична строке формата функции `sprintf()`, поэтому мы ее не будем рассматривать. Если вы не знакомы с языком C (может быть и такое, хотя это маловероятно), прочитать о строке формата можно по адресу:

http://www.tcl.tk/man/tcl/TclCmd/format.htm

Рассмотрим вывод форматированной строки:

```
set fmt "Здесь будет строка %s, а здесь — число %d"
puts [format $fmt "Hello" 123456]
```

Сначала мы присваиваем переменной `fmt` значение строки формата, которая содержит модификаторы формата `%s` и `%d`. Затем командой `puts` выводим строку, отформатированную с помощью команды `format`. Команде `format` мы передаем строку формата и два аргумента — строку и число.

Как обычно, при выводе строк можно использовать символы `\n`, `\t`, `\a` (звуковой сигнал) и др., которые вы привыкли использовать при программировании на C.

24.1.3. Группировка аргументов

В табл. 23.1 было сказано, что двойные кавычки и фигурные скобки используются для группировки аргументов. Представим, что вы объявили следующие переменные:

```
set val1 "Value 1";
set val2 "Value 2";
set val3 "Value 3";
```

Теперь выполним две команды `puts`:

```
puts "$val1 $val2 $val3";
puts {$val1 $val2 $val3};
```

Первая команда выведет строку:

```
Value 1 Value 2 Value 3
```

Вторая команда выведет другую строку:

```
val1 val2 val3
```

Как видите, в первом случае произошла подстановка значений переменных, а во втором случае — нет.

24.1.4. Переменные

Как вы уже успели заметить, значения переменным присваиваются с помощью команды `set`. Эта команда заменяет оператор присваивания (`=`), другого способа присвоить переменной значение нет.

Синтаксис команды `set` следующий:

```
set имя_переменной значение
```

Переменная может быть обычной скалярной переменной или массивом. Рассмотрим несколько примеров использования команды `set`:

```
set a 123456
set s1 String1
set s2 "String 2"
set s3 {String 3}
```

Первая команда просто присваивает переменной `a` число 123 456. Как видите, вам не нужно указывать тип переменной — он определяется интерпретатором автоматически, в зависимости от присваиваемого значения.

Если присваиваемая строка состоит из одного слова, можно ее не заключать в кавычки или фигурные скобки. Однако лучше всегда использовать фигурные скобки или кавычки для большей однозначности.

Посмотрите на все четыре команды: после них нет точки с запятой. Это не ошибка — ведь команды могут разделяться символом новой строки. Но точку с запятой я не указывал намеренно. Посмотрите на следующие две команды:

```
set s1 String1;
set s2 {String2};
```

Переменной `s2` будет присвоено значение `String2` — это очевидно. А вот какое значение будет присвоено переменной `s1` — не ясно: либо `String1;`, либо `String1`. Нужно вызвать интерпретатор `tclsh`, чтобы проверить это. Давайте договоримся: чтобы наш код был однозначным и понятным, строки будем заключать либо в кавычки, либо в фигурные скобки, а после каждой команды будем указывать точку с запятой. Позже, когда вы будете уверенно владеть синтаксисом TCL, от точки с запятой можно отказаться (что и будет продемонстрировано далее), но тогда не забывайте о том, что в одной строке должна быть одна и только одна TCL-команда.

Для присваивания переменной результата выполнения какой-нибудь команды нужно использовать фигурные скобки. Команда `expr` вычисляет значение арифметического выражения, например:

```
set a [expr 2 + 2];
```

Символ `$` используется для подстановки значения переменной:

```
set b [expr $a + 2];
```

Переменной `a` будет присвоено значение 4 (2 + 2), а переменной `b` — 6 (4 + 2). Если вам нужно вывести сам символ `$`, то укажите слэш перед ним:

```
puts \$msg;
```

Вот небольшой пример вычисления случайного числа в диапазоне от 0 до 100:

```
set min 1
set max 100
set random [expr {round($min + (rand() * ($max – $min)))}]
puts "Случайное число: $random"
```

24.1.5. Процедуры

В любом языке программирования можно создавать подпрограммы. В TCL подпрограммы называются процедурами. Объявить процедуру можно так:

```
proc имя_процедуры { список_аргументов } {
  ... ...
}
```

Рассмотрим небольшой пример: процедура, удваивающая переданный ей аргумент.

```
proc dbl {arg} {
set result [expr $arg * 2];
return result;
}
```

Если процедуре нужно передать несколько аргументов, то их указывают через пробел:

```
proc mysum {arg1 arg2} {
set result [expr $arg1 + $arg2];
return result;
}
```

Использовать процедуры можно так:

```
set a [expr dbl(3)];
set b [expr mysum(2,5)];
```

24.1.6. Получаем ввод пользователя

С помощью команды `gets` можно получить строку со стандартного ввода, из файла, сетевого сокета. Но в этой главе мы будем рассматривать получение данных только со стандартного ввода. Синтаксис команды `gets` следующий:

```
gets канал имя-переменной
```

Команда `gets` читает строку, пока не будет достигнут символ конца строки (EOL). Именно поэтому пока мы рассматриваем чтение только со стандартного ввода, когда нажатие клавиши <Enter> расценивается как конец строки. А о работе с файлами, когда нужно будет учитывать конец файла (EOF), мы поговорим в *главе 25*.

Рассмотрим небольшой пример:

```
puts -nonewline "Введите что-нибудь: "
flush stdout
set count [gets stdin user_input]
puts "Вы ввели: $user_input"
puts "Во введенной вами строке $count символов"
```

Сначала мы выводим приглашение для ввода текста. Затем мы сбрасываем буферы стандартного вывода, хотя обычно это можно и не делать, зато мы будем точно знать, что наше приглашение было выведено на экран.

Команда `gets` читает строку со стандартного ввода (`stdin`) в переменную `user_input` и возвращает целое число — количество символов в прочитанной строке. Это число мы присваиваем переменной `count`. Затем мы выводим переменные `user_input` и `count`.

24.1.7. Математические операции

Для выполнения математических операций используется команда `expr`. Синтаксис ее следующий:

```
expr аргумент1 аргумент2 ...
```

Каждый аргумент может быть либо оператором, либо операндом, например:

```
expr 2 + 3
```

В этом случае команде `expr` мы передаем три аргумента: операнд 2, оператор + и операнд 3. Команда возвращает результат вычисления арифметической операции. В табл. 24.2 приводится список операторов. В принципе, операторы подобны аналогичным использующимся в других языках программирования. Операторы в табл. 24.2 приводятся в порядке уменьшения приоритета.

Таблица 24.2. *Математические операторы TCL*

Приоритет	Оператор	Целое	Вещественное	Строка	Описание
1	-	+	+	-	Унарный минус
1	+	+	+	-	Унарный плюс
1	~	+	-	-	Побитное NOT
1	!	+	+	-	Логическое NOT
2	*	+	+	-	Умножение
2	/	+	+	+	Деление
2	%	+	-	-	Остаток
3	+	+	+	-	Сложение
3	-	+	+	-	Вычитание
4	<<	+	-	-	Сдвиг влево
4	>>	+	-	-	Сдвиг вправо
5	<	+	+	-	Меньше чем
5	>	+	+	-	Больше чем
5	<=	+	+	-	Меньше или равно
5	>=	+	+	-	Больше или равно
6	==	+	+	-	Равно
6	!=	+	+	-	Не равно
7	eq	-	-	+	Равно

Таблица 24.2 *(окончание)*

Приоритет	Оператор	Целое	Вещественное	Строка	Описание
7	ne	-	-	+	Не равно
8	$	+	-	-	Побитное AND
9	^	+	-	-	Побитное исключительное OR
10	|	+	-	-	Побитное OR
11	&&	+	-	-	Логическое AND
12	||	+	-	-	Логическое OR
13	x ? y : z	+	+	+	If .. then .. else

В таблице помимо наименования самого оператора, его приоритета и описания приводятся типы допустимых операндов. Например, операторы == и != используются только для сравнения числовых значений, но не строк. Для сравнения строк нужно использовать операторы eq и ne.

24.1.8. Условная команда *if*

В любом языке программирования есть условный оператор. Поскольку в TCL операторы называют командами, то у нас есть не условный оператор, а условная команда if.

В простейшем случае команда if выглядит так:

```
if { Логическое_выражение } {
    Команды
}
```

Если логическое выражение истинно, то будут выполнены команды. Пример условной команды:

```
if {$a < 0} {
    set a 0
}
```

Полный синтаксис условной команды выглядит так:

```
if {выражение1} ?then? {
команды1
} elseif {выражение2} ?then? {
команды2
} elseif
...
else {
командыN
}
```

Служебное слово `then` можно игнорировать, что немного сокращает программный код и делает синтаксис больше похожим на C.

24.1.9. Команда *while*

Цикл `while` можно организовать командой `while`:

```
while {условие} {тело_цикла}
```

Если условие истинно, будут выполнены команды из тела цикла. Записывать цикл `while` в одну строку, как и команду `if`, неудобно, поэтому целесообразно (для улучшения читабельности программы) записывать цикл в несколько строк:

```
while { $a < 10 } {
    puts $a
    incr a
}
```

Простейший цикл, выполняющийся до тех пор, пока переменная `a` меньше 10. В теле цикла происходит вывод значения переменной `a` и ее увеличение на 1 командой `incr`.

24.1.10. Команда *for*

Синтаксис команды `for` следующий:

```
for {start} {test} {next} {body}
```

Синтаксис команды `for` напоминает синтаксис оператора `for` в любом языке программирования. Сначала выполняется команда `start`, обычно она используется для инициализации счетчика цикла. Затем следует условие `test`, которое определяет, будут ли выполнены команды, указанные в `body`. А команды, указанные в `next`, будут выполнены после итерации цикла. Обычно `next` содержит команду увеличения счетчика цикла.

Рассмотрим пример:

```
for {set i 1} {$i < 10} {incr i} {
puts $i;
}
```

Мы только что познакомились с основами синтаксиса языка TCL, настало время перейти к более сложным "материям" — к строкам и массивам.

24.2. Строки

24.2.1. Команда *string*

Строки в любом языке программирования заслуживают отдельного разговора, поэтому им отведен целый параграф. Знакомство со строками начнем с команды

string, с помощью которой можно выполнить практически все операции над строками:

```
string опция аргумент1 [аргумент2 аргумент 3 ...]
```

Команде string нужно передать опцию и хотя бы один аргумент. Количество аргументов зависит от выбранной вами опции. Например, если вы выбрали опцию compare, которая используется для сравнения строк, то аргумент никак не может быть один. Опция compare выполняет сравнение двух строк, поэтому и аргументов должно быть два. В табл. 24.3 представлены опции команды string.

Таблица 24.3. *Опции команды* `string`

Опция	Описание
bytelength	Возвращает число байтов, необходимых для хранения строки в памяти
compare	Сравнивает две строки (лексикографический порядок)
equal	Возвращает 1, если строки лексикографически идентичны, и 0, если это не так
first	Возвращает индекс первого вхождения подстроки в строку
index	Возвращает символ, который находится в указанной позиции строки
is	Проверяет, принадлежит ли заданная строка к указанному классу строк
last	Возвращает индекс последнего вхождения подстроки в строку
length	Возвращает количество символов в строке
map	Заменяет подстроки новыми значениями
match	Проверяет, есть ли в строке последовательность символов, соответствующая образцу
range	Возвращает подстроку, заданную параметрами start и end
repeat	Повторяет строку указанное количество раз
replace	Производит замену в строке
tolower	Переводит все символы строки в нижний регистр
toupper	Переводит все символы строки в верхний регистр
totitle	Переводит первый символ строки в верхний регистр
trim	Удаляет первый и последний символы последовательности, которая соответствует заданному шаблону
trimleft	Удаляет первый символ последовательности, которая соответствует заданному шаблону
trimright	Удаляет последний символ последовательности, которая соответствует заданному шаблону

Теперь рассмотрим базовые операции над строками.

24.2.2. Сравнение строк

Для сравнения строк нужно использовать либо операторы `eq`, `ne` условной команды `if`, либо опции `compare`, `equal` команды `string`:

```
set s1 = "string";
set s2 = "string";
if {$s1 eq $s2} {
puts {Строки s1 и s2 равны};
} else {
puts {Строки s1 не s2 равны};
}
```

Теперь рассмотрим использование опции `compare`:

```
string compare [-nocase] [-length N] strng1 string2
```

Мы будем сравнивать строки, заданные параметрами `string1` и `string2`. По умолчанию при сравнении строк учитывается регистр символов. Чтобы отключить проверку регистра, используйте необязательную опцию `-nocase`. А если вам нужно сравнить первые *N* символов обеих строк — используйте опцию `-length N`.

Опция `compare` работает так же, как и C-функция `strcmp()`, поэтому она возвращает такие же значения:

- –1 — если строка `string1` лексикографически меньше строки `string2`;
- 1 — если строка `string1` лексикографически больше строки `string2`;
- 0 — если строки эквивалентны.

Рассмотрим пример использования `compare`:

```
puts -nonewline "Введите имя пользователя: ";
flush stdout;
get stdin str;
if{![string compare $str "admin"]}{
puts "Вы не можете использовать имя admin";
} else {
puts "Вы ввели имя $str";
}
```

Опция `equal` тоже используется для сравнения двух строк (ее синтаксис аналогичен опции `compare`). Разница между ними заключается в том, что `equal` требует полной идентичности строк и возвращает 1 (true), если строки идентичны, и 0, если строки неидентичны. Для обычного сравнения строк удобнее использовать опцию `equal`, а для всевозможных алгоритмов сортировки, где важно знать, какая из строк является лексикографически меньше или больше — опцию `compare`. Рассмотрим тот же пример с проверкой имени пользователя, но написанный с использованием опции `equal`:

```
puts -nonewline "Введите имя пользователя: ";
flush stdout;
get stdin str;
```

```
if{[string equal $str "admin"]}{
puts "Вы не можете использовать имя admin";
} else {
puts "Вы ввели имя $str";
}
```

Опция `match` возвращает 1, если строка соответствует заданному шаблону. Синтаксис опции `match` следующий:

```
string match [-nocase] шаблон строка
```

Параметр `-nocase` отключает проверку регистра символов и не является обязательным. Шаблон может содержать символы `*` и `?`. Первый заменяет последовательность символов любой длины, а второй — только один произвольный символ.

```
if{[string match "*.jpg" $filename]}{
puts "JPEG-изображение";
} else {
puts "Другой файл";
}
```

Кроме символов `*` и `?` вы также можете указать диапазон символов:

```
if {[string match {[A-Z]} $filename]} {
# команды 1
} else {
# команды 2
}
```

24.2.3. Получаем информацию о строках

Сравнение — далеко не единственная операция, которая может понадобиться при работе со строками. Часто возникает потребность определить, сколько символов в строке, есть ли в строке та или иная подстрока и т. д.

Начнем с опций `length` и `bytelength`:

```
string length строка
string bytelength строка
```

Разница между этими двумя опциями огромна. Первая возвращает длину строки в символах, а вторая — в байтах. В самом простом случае один символ равен одному байту, но не забывайте, что TCL использует Unicode, где для представления одного символа требуется три байта памяти.

Небольшой пример:

```
set str "©";
puts "В строке [string length $str] символов";
puts "Строка занимает [string bytelength $str] байтов";
```

Опция `index` возвращает символ, находящийся в строке на позиции `N`:

```
string index строка N
```

Нумерация символов начинается с 0. Если у нас есть строка "Привет", то символ с номером *N* — это "П", с номером 1 — "р" и т. д.

Опции `first` и `last` используются для поиска первого и последнего вхождения подстроки в строку:

```
string first подстрока строка [start]
string last подстрока строка [end]
```

Опция `first` ищет первое вхождение подстроки в строку и возвращает индекс первого символа подстроки. Опция `last`, аналогично, возвращает индекс первого символа последнего вхождения подстроки в строку. Необязательные аргументы `start` и `end` позволяют указать начальный и конечный индексы символа, с которого нужно начать поиск. По умолчанию поиск начинается с индекса 0.

Пример использования опций `first` и `last`:

```
set str "1234 ## 5678 ## 999";
set start [string first "##" $str];
set end [string last "##" $str];
puts "start = $start";              # Выведет 5
puts "end = $end";                  # Выведет 13
```

Опция `range` позволяет получить подстроку, начальный и конечный символы которой заданы параметрами `start` и `end`:

```
string range строка start end
```

Думаю, опция `range` понятна без дополнительных примеров, поэтому перейдем сразу к опции `replace`, выполняющей поиск и замену в строке. Синтаксис опции `replace` следующий:

```
string replace строка start end [новая_строка]
```

Опция `replace` удаляет из строки подстроку, заданную границами `start` и `end`. Если указан необязательный параметр `newstr`, удаленный текст будет заменен значением параметра `новая_строка`. Пример:

```
set s "Привет";
set new_s [string replace $s 0 5 "Hello"];
puts $new_s;  # Выведет Hello
```

Следующая опция — очень полезна. Опция `is` позволяет проверить, является ли строка числом, обычным символом, пробельным символом и т. д. Синтаксис выглядит так:

```
string is класс [-strict] строка
```

Возвращает 1, если строка принадлежит к классу, указанному параметром `класс`, 0 — в противном случае. Пустая строка (`""`) относится ко всем классам, если не задан параметр `-strict` (в этом случае пустая строка не относится ни к одному из классов). Классы строк (значения параметра класс) указаны в табл. 24.4.

Пример:

```
if {[string is integer "123"]} { puts "Interger!" }
```

Таблица 24.4. *Классы строк*

Класс	Описание
alnum	Любой алфавитный (Unicode) символ или цифра
alpha	Любой алфавитный символ
ascii	Любой символ из 7-битного набора ASCII
boolean	Любая форма, используемая для представления логических значений
control	Любой управляющий символ (из кодировки Unicode)
digit	Любая цифра (Unicode)
double	Любая форма, используемая для представления вещественных значений (позволяет проверить, является ли строка вещественным числом)
false	Любая форма, используемая для представления логического значения FALSE
graph	Любой печатаемый символ (Unicode), кроме пробела
integer	Позволяет проверить, является ли строка целым числом
lower	Позволяет проверить, все ли символы строки находятся в нижнем регистре (Unicode)
print	Любой печатаемый символ (Unicode), в том числе пробел
space	Любой пробельный символ (Unicode)
true	Любая форма, используемая для представления логического TRUE
upper	Позволяет проверить, все ли символы строки находятся в верхнем регистре (Unicode)
wordchar	Любой словесный символ (Unicode) — символ, который может использоваться в слове
xdigit	Шестнадцатеричная цифра

24.2.4. Модификация строк

К опциям модификации строк относят опцию повторения символов, опции модификации регистра символов, опции удаления символов. Начнем с опции repeat, повторяющей заданную строку N раз:

```
string repeat строка N
```

Пример:

```
puts [string repeat "X" 10]
```

Данная команда выведет символ X десять раз.

К опциям изменения регистра символов относятся опции, описанные в табл. 24.3. Синтаксис опций следующий:

```
string toupper строка [start] [end]
string tolower строка [start] [end]
string totitle строка [start] [end]
```

Необязательные параметры start и end задают начало и конец строки, если значения по умолчанию вас не устраивают (по умолчанию start = 0, end = длина_строки – 1).

Опции trimleft, trimright и trim (см. табл. 24.3) удаляют "лишние" символы в строке:

```
string trimleft строка [chars]
string trimright строка [chars]
string trim строка [chars]
```

Необязательный параметр chars задает один или несколько символов, которые должны быть удалены из строки. По умолчанию удаляются все пробельные символы (пробелы, символы новой строки, символы перевода каретки, символы табуляции).

24.2.5. Конкатенация строк

Рассмотрим еще одну операцию — конкатенацию, или слияние строк. Слияние строк легко выполнить с помощью команды set:

```
set str1 ", world";
set str2 "Hello $str1";
puts $str2;
```

Все бы хорошо, но такую конструкцию неудобно использовать в цикле. В этом случае гораздо удобнее использовать команду append:

```
append переменная значение1 [... значениеN]
```

Команда append добавляет в конец строки, заданной переменной, указанное значение (или список значений — в порядке их указания):

```
append $s "Hello " "world " "again";
puts $s;
```

24.3. Списки

24.3.1. Команда *list*: создание списка

Список — это упорядоченная последовательность значений, разделенных пробелом. Для создания списка используется команда list:

```
list элемент1 ... элементN
```

Обычно list используется в паре с set:

```
set cars [list VAZ GAZ ZAZ]
```

Мы создали список cars с элементами VAZ, GAZ и ZAZ. Для большей определенности желательно строковые элементы списка заключать в кавычки:

```
set cars [list "VAZ" "GAZ" "ZAZ"]
```

При использовании кавычек (или фигурных скобок) мы можем создавать элементы списка, содержащие пробелы:

```
set cars [list "VAZ" "GAZ" "ZAZ" "Alfa Romeo"]
```

24.3.2. Команда *concat*: слияние списков

Команда `concat` используется для объединения списков. Элементы в новом списке будут следовать в порядке, в котором указаны списки в команде `concat`. Пример:

```
set cars1 [list "VAZ" "GAZ" "ZAZ" "Alfa Romeo"];
set cars2 [list "Mazda" "Toyota" "Nissan"];
set allcars [concat $cars1 $cars2];
```

Список `allcars` будет выглядеть так:

```
"VAZ" "GAZ" "ZAZ" "Alfa Romeo" "Mazda" "Toyota" "Nissan"
```

24.3.3. Команда *lappend*: добавление элемента в конец списка

Если вы желаете добавить элемент в конец какого-нибудь списка, проще всего это сделать с помощью команды `lappend`:

```
lappend cars1 "VW"
```

24.3.4. Доступ к элементам списка

Команда `lindex` предоставляет доступ к элементам списка:

```
lindex переменная [N]
```

Здесь первый параметр — это переменная-список, второй — номер элемента. Если номер элемента не задан, возвращается весь список. Рассмотрим код, выводящий первые три элемента списка (нумерация элементов начинается с 0):

```
for {set i 0} {$i < 3} {incr i} {
puts [lindex $cars $i];
}
```

Понятно, что наша программа была бы более универсальна, если бы мы знали длину списка. Для этого используется команда `llength`:

```
for {set i 0} {$i < [llength $cars]} {incr i} {
puts [lindex $cars $i];
}
```

Получить диапазон списка можно командой `lrange`:

```
lrange список start end
```

Команда возвращает список, состоящий из элементов указанного списка с индексами от `start` до `end`.

24.3.5. Вставка новых элементов

Команда `append` может добавить элемент только в конец списка. Команда `linsert` позволяет вставить новый элемент в заданную позицию:

```
linsert список индекс элемент1 [элемент2 ... элементN]
```

Первый аргумент — переменная списка, в который будут добавлены новые элементы (или только один новый элемент). Новые аргументы будут добавлены перед элементом с указанным индексом. Команда `linsert` возвращает новый список, исходный список не модифицируется. Если индекс меньше 0, новые элементы будут вставлены в начало списка. Если индекс превышает количество элементов в списке, тогда новые элементы будут вставлены в конец списка.

24.3.6. Замена и удаление элементов списка

Для замены одного или нескольких элементов списка используется команда `lreplace`:

```
lreplace список start end [элемент...]
```

Команда возвращает новый список, созданный путем замены элемента или элементов с индексами от `start` до `end` новыми элементами (или элементом), указанными после `end`. Если элементы после `end` не указаны, элементы от `start` до `end` будут удалены из списка.

Если `start` меньше 0, считается, что `start` = 0. Если `end` меньше, чем `start`, все указанные элементы будут вставлены в начало списка без замены существующих элементов списка. Если список пуст, все указанные элементы будут добавлены в список.

24.3.7. Поиск элемента

Найти элемент списка, соответствующий шаблону, можно с помощью команды `lsearch`:

```
lsearch [опция] список шаблон
```

Каждый элемент указанного списка будет сопоставлен с шаблоном. Команда возвращает индекс первого элемента, соответствующего шаблону. Если совпадений не найдено, возвращается –1.

Опции позволяют уточнить тип поиска или тип шаблона. В большинстве случаев вы или вообще не будете указывать какие-либо опции, или же воспользуетесь опцией `-regexp`, сообщающей `lsearch`, что шаблон представляет собой регулярное выражение:

```
lsearch -regexp список выражение
```

Еще одна полезная опция — `-start N`, позволяющая задать номер элемента, с которого следует начать поиск. Она очень полезна, когда нужно найти все элементы, соответствующие шаблону.

В этом случае алгоритм будет следующим:

1. Найти первый элемент, соответствующий шаблону, запомнить его номер.
2. Запустить снова поиск элементов с параметром `-start N+1`, где `N` — номер уже найденного элемента.
3. Запомнить снова номер найденного элемента и опять запустить поиск с опцией `-start N+1`, где `N` — номер последнего найденного элемента и т. д., пока не будет достигнут конец списка.

Пример поиска всех совпадений в списке:

```
set cars [list "GAZ-21" "VAZ" "GAZ" "ZAZ" "GAZ-24"];
set n 0;
set first 0;
while { $n>-1} {
if { $first > 0 } {
set n [lsearch -start $n+1 $cars "GAZ*"];
} else {
set n [lsearch -start $n $cars "GAZ*"];
if { $n > -1 } incr first;
}
if { $n > -1 } { puts "Find $n"; }
}
```

Программа выводит номера элементов списка, совпадающих с шаблоном `"GAZ*"`:

```
Find 0
Find 2
Find 4
```

Рассмотрим подробнее нашу программу поиска всех совпадений. Переменная `n` — это индекс найденного элемента. Переменная `first` — признак того, что найдено первое совпадение. До того как найдено первое совпадение, мы должны производить поиск так:

```
set n [lsearch -start $n $cars "GAZ*"];
```

При этом переменная `n` должна быть равна 0. Как только найдено первое совпадение, поиск должен происходить так:

```
set n [lsearch -start $n+1 $cars "GAZ*"];
```

Если в качестве параметра `start` указать индекс `n` (т. е. номер уже найденного элемента), то программа опять найдет этот же элемент, и так будет продолжаться до бесконечности — программа зациклится.

Если `first = 0`, то первый элемент еще не найден. Как только мы найдем первый элемент, мы увеличим значение `first`.

Последний вызов `lsearch` возвратит –1: будет достигнут конец списка. Чтобы препятствовать выводу `Find -1`, мы перед выводом номера найденного элемента проверяем переменную `n`: сообщение будет выведено, если `n > -1`.

Может это и не самый лучший алгоритм поиска, зато он нормально реагирует на 0-й элемент и "проходит" по всему списку.

А вот алгоритм с логической ошибкой:

```
set cars [list "VAZ" "GAZ" "ZAZ" "GAZ-24"];
set n 0;
while { $n>-1} {
set n [lsearch -start $n+1 $cars "GAZ*"];
if { $n > -1 } { puts "Find $n"; }
}
```

Программа будет прекрасно работать, пока первый элемент не совпадает с шаблоном. Измените первый элемент списка — вместо элемента `VAZ` установите `GAZ` и вы увидите, что программа выведет только номера 1 и 3, но не элемент с номером 0, поскольку поиск сразу начинается с элемента `n + 1` (`0 + 1`).

24.3.8. Сортировка списка

Отсортировать список можно с помощью команды `lsort`:

```
lsort [опция] список
```

Опции позволяют задать порядок сортировки и установить параметры сортировки (табл. 24.5).

Таблица 24.5. *Опции* `lsort`

Опция	Описание
`-ascii`	Элементы списка считаются Unicode-строками
`-dictionary`	Сортировка в словарном стиле
`-integer`	Считать все элементы списка целыми числами
`-real`	Считать все элементы списка вещественными числами
`-command команда`	Выполнить TCL-команду для сравнения элементов (так называемая пользовательская сортировка)
`-increasing`	Сортировать в порядке возрастания
`-decreasing`	Сортировать в порядке убывания
`-index индекс`	Начать сортировку с элемента с указанным индексом
`-unique`	Удаляет дублирующиеся элементы

24.3.9. Преобразование строки в список и обратно

Язык TCL позволяет преобразовать строку в список и обратно, т. е. список в строку. Для преобразования строки в список используется команда `split`:

```
split строка [разделитель]
```

Указанная строка будет преобразована в список, элементами списка будут слова — последовательности символов, разделенных пробелом. По умолчанию в качестве разделителя используется пробел, но вы можете указать любой другой символ (например, при обработке формата CSV нужно указать двоеточие) и даже несколько символов.

Пример:

```
set words [split $sentence]
```

Разбивает предложение из переменной `$sentence` в список `words`.

Для обратного преобразования, т. е. для преобразования списка в строку, используется команда `join`:

```
join список [разделитель]
```

По умолчанию будет создана строка, состоящая из элементов списка, разделенных пробелами, но вы можете указать другой разделитель:

```
set $sentence2 [join $words]
```

24.3.10. Цикл *foreach*

Для "путешествия" по списку можно использовать любой из циклов — `for` или `while`, но гораздо удобнее использовать для этого цикл `foreach`. Аналогичный цикл есть в PHP, и его использование гораздо удобнее использования цикла `for`. Цикл `foreach` "проходит" по всем элементам списка, вам не нужно вычислять число элементов, вам нужно просто указать имя списка.

Синтаксис `foreach`:

```
foreach переменная список { тело }
```

Здесь: `список` — список, который мы будем исследовать, `переменная` — имя переменной, которую можно использовать для доступа к элементу цикла; `тело` — команды, которые будут выполнены для обработки списка.

Рассмотрим вывод списка обычным циклом `for`:

```
for {set i 0} {$i < [llength $cars]} {incr i} {
puts [lindex $cars $i];
}
```

А теперь выполним те же действия, но с помощью `foreach`:

```
foreach item $cars {
       puts $item;
}
```

Как видите, использование `foreach` гораздо удобнее обычного цикла `for`.

> ***Примечание***
>
> В других языках программирования имеется инструкция для прерывания цикла — `break`. Аналогичная команда есть и в TCL: она также называется `break`, и вы можете использовать ее в своих программах. Также имеется команда `continue`, прерывающая текущую итерацию цикла.

24.4. Массивы

24.4.1. Отличие массивов от списков

По большому счету, массивы и списки — практически одно и то же. Просто список — это частный вид массива. И список, и массив предполагает доступ к элементу по индексу, но у списка индекс может быть только целым числом, а у массива в качестве индекса может выступать строка. Учитывая этот факт, массив является более универсальным средством обработки данных.

Пусть у нас есть два пользователя: admin и guest. Создадим массив `password`, содержащий пароли пользователей:

```
set passwords(admin) "qwerty";
set passwords(guest) "123";
```

Получить доступ к элементу массива можно так:

```
puts $passwords(admin);
```

24.4.2. Команда *array*: обработка массивов

Подобно команде `string`, для работы с массивами используется команда `array`, у которой есть свой набор опций (табл. 24.6). В зависимости от опции формат вызова команды `array` может отличаться.

Таблица 24.6. *Опции команды* `array`

Команда	Описание
`array anymore массив ключ_перебора`	Возвращает 1, если есть элементы для перебора командой `array nextelement`, в противном случае возвращается 0
`array nextelement массив ключ_перебора`	Возвращает индекс следующего элемента. Если в массиве больше нет элементов, возвращается пустая строка
`array startsearch массив`	Подготавливает массив для поиска и возвращает ключ перебора, который нужно использовать в командах `anymore` и `nextelement`
`array donesearch массив ключ_перебора`	Уничтожает информацию о поиске, эту команду нужно выполнять, когда уже нашли, что искали
`array exist массив`	Возвращает 1, если массив существует (переменная с указанным именем существует и является массивом), 0 — в противном случае (когда переменная не существует или является обычной скалярной переменной)
`array get массив`	Возвращает список, в котором на нечетных местах (1, 3 и т. д.) индексы массива, а на четных — соответствующие им значения

Таблица 24.6 *(окончание)*

Команда	Описание
`array set массив список`	Инициализирует массив элементами списка. Список должен быть в формате команды `array get`, т. е. на нечетных местах — индексы массива, на четных — значения
`array size массив`	Возвращает количество элементов массива
`array names массив [шаблон]`	Возвращает список индексов для ассоциативного массива. Если задан шаблон, то будут возвращены только индексы элементов, соответствующие шаблону

Рассмотрим несколько примеров работы с массивом:

```
array set users [list \
{1} {Иван Иванов} \
{2} {Петр Петров} \
{3} {Сергей Сидоров}];

puts "В массиве users [array size users] элементов";
puts "Идентификатор 3 соответствует элементу: $users(3)";
```

Перебрать элементы массива можно так:

```
foreach id [array names users] {
       puts "$users($id) индекс = $id";
}
```

24.5. Ошибки начинающих TCL-программистов

Всегда считается, что учиться с нуля намного проще, чем переучиваться — делаешь меньше ошибок. Но реальность немного иная. Вряд ли TCL станет вашим первым языком программирования. Скорее всего, вы уже знаете C и еще несколько языков программирования, вполне возможно Basic, Pascal, PHP, Java. Учитывая, что вы знаете хотя бы несколько языков из этого списка (например, C и PHP), при программировании на TCL вы будете делать много ошибок.

Чаще всего ошибки связаны с неправильным использованием символа `$`, выполняющего подстановку значения переменной. В языке C вообще не нужно указывать символ `$` перед именем переменной, поскольку прежде чем использовать переменную, вы ее объявляете. В PHP предварительного объявления нет, но символ `$` используется перед каждым именем переменной — вне зависимости от ситуации: присваиваете ли вы значение переменной или передаете переменную функции.

В TCL, когда вы присваиваете значение переменной, нужно указать просто ее имя, но когда вам нужно "добраться" к значению переменной, нужно указать символ `$`. Рассмотрим небольшой пример:

```
set s1 = "string";
set s2 = "string";
if {s1 eq s2} {
puts {Строки s1 и s2 равны};
} else {
puts {Строки s1 не s2 равны};
}
```

Вместо вывода вполне ожидаемой строки (о том, что `s1` и `s2` равны) вы получите сообщение об ошибке. Почему? Да потому что вы забыли указать символ `$` перед названием переменных. Но это еще ничего, т. к. TCL контролирует ситуацию и сообщит, что перед именем переменной нужно указывать `$`. Совсем другое дело — использование команды `string`, где аргументами опций могут быть как переменные, так и символьные константы:

```
set s "Privet";
set new_s [string replace s 1 5 "Hello"];
puts $new_s;
```

Очевидно, вы рассчитываете получить строку "PHello", а вместо этого программа выведет строку `s`. А вот если добавить `$` перед `s`, то все станет на свои места:

```
set new_s [string replace $s 1 5 "Hello"];
```

Интерпретатор не сообщит об ошибке, вы просто получите не тот результат. В простых программах найти и выследить ошибку легко, но в сложных программах, которые будут обрабатывать большие массивы данных, определить, в чем ошибка, порой очень трудно.

Следующая ошибка начинающего TCL-программиста заключается в использовании круглых скобок команды `if` вместо фигурных:

```
if ($new_s eq "Hello") { puts "что-то вывести" }
```

Да, с круглыми скобками команда `if` становится похожей на аналогичный условный оператор в других языках программирования, но, увы, использование круглых скобок является нарушением синтаксиса. Правильный вариант:

```
if {$new_s eq "Hello"} { puts "что-то вывести" }
```

Еще одна ошибка связана с обилием кавычек: иногда программист забывает напечатать пробел между `if` и фигурной кавычкой, в результате TCL не может выполнить команду. Помните, что TCL-программа — это список команд с аргументами, а команда и аргументы должны разделяться пробелами.

ГЛАВА 25

Работа с файлами

25.1. Открываем и закрываем файлы

Для открытия файла используется команда `open`:

```
open имя_файла [режим_доступа] [права_доступа]
```

Первый параметр — это имя файла, который нужно открыть. Второй параметр — режим доступа (табл. 25.1). Он определяет, какой доступ вы хотите получить к файлу: чтение, запись и т. д. Значения этого параметра похожи на значения аналогичного параметра С-функции `fopen()`.

Таблица 25.1. *Режим доступа к файлу*

Режим	Описание
r	Только чтение. Открываемый файл должен существовать
r+	Чтение и запись. Открываемый файл должен существовать
w	Только запись. Если файл существует, он будет перезаписан; если файл не существует, будет создан
w+	Чтение и запись. Если файл существует, он будет перезаписан; если файл не существует, будет создан
a	Дозапись в конец файла. Оптимально для журналов. Создает файл, если он не существует
a+	Чтение и запись (в конец файла). Создает файл, если он не существует

Третий параметр задает права доступа в UNIX-стиле. Права доступа нужно указывать, когда вы создаете файл. Данный параметр игнорируется, если вы используете режим "только чтение".

Команда `open` возвращает идентификатор канала, который мы будем применять в функциях чтения/записи файла. Вы можете использовать стандартные идентифика-

торы: stdout (стандартный вывод), stdin (стандартный ввод) и stderr (стандартный поток ошибок).

Закрыть файл можно с помощью команды close, которой нужно передать идентификатор канала:

```
close id
```

Пример открытия и закрытия файла:

```
set fId [open test.txt r];
puts "Файл test.txt открыт, ID канала $fId";
close $fId;
```

Обработать ошибку при открытии файла можно так:

```
if {[catch {set fId [open test.txt r+]} err]} {
puts "не могу открыть файл, ошибка: $err";
return 1;
} else {
puts "Файл test.txt открыт, ID канала $fId";
close $fId;
}
```

25.2. Чтение файла

Прочитать содержимое файла можно одной из трех команд:

- gets — подходит для построчного чтения текстовых файлов;
- read — читает данные блоками, не обращая внимания на маркеры EOL (End Of Line, конец строки), пока не будет достигнут конец файла. Подходит как для чтения текстовых, так и для чтения двоичных файлов;
- scan — для чтения форматированного ввода.

Сначала рассмотрим команду gets. Напишем программу, читающую ввод пользователя и выводящую прочитанную строку на экран:

```
set line "";
gets stdin line;
puts $line;
```

Первая строка инициализирует переменную line, вторая — читает строку со стандартного ввода, третья — выводит прочитанную строку на экран.

Теперь усложним задачу. Напишем программу, которая будет построчно читать и выводить строки из текстового файла test.txt. Перед выводом каждой строки программа будет подсчитывать и выводить количество символов в этой строке, а затем сообщит, сколько всего символов прочитано и сколько всего строк в файле. Для подсчета количества символов будем использовать команду gets: она возвращает количество прочитанных символов или –1 в случае ошибки (например, если прочитан весь файл). Код программы get.tcl представлен в листинге 25.1.

Листинг 25.1. Программа get.tcl

```
set fId [open test.txt r];
set tChars 0;
set tLines 0;

while {[set count [gets $fId line]] != -1} {
       puts "($count символов) $line"
       incr tChars $count
       incr tLines
}

puts "Всего $tChars символов"
puts "Всего $tLines строк"

close $fId
```

В каждой итерации цикла `while` мы увеличиваем "общие" счетчики `tChars` (общее количество символов) и `tLines` (общее количество строк в файле). Результат работы программы представлен на рис. 25.1.

Теперь рассмотрим чтение файла с помощью `read`:

```
read id_канала [количество_символов]
```

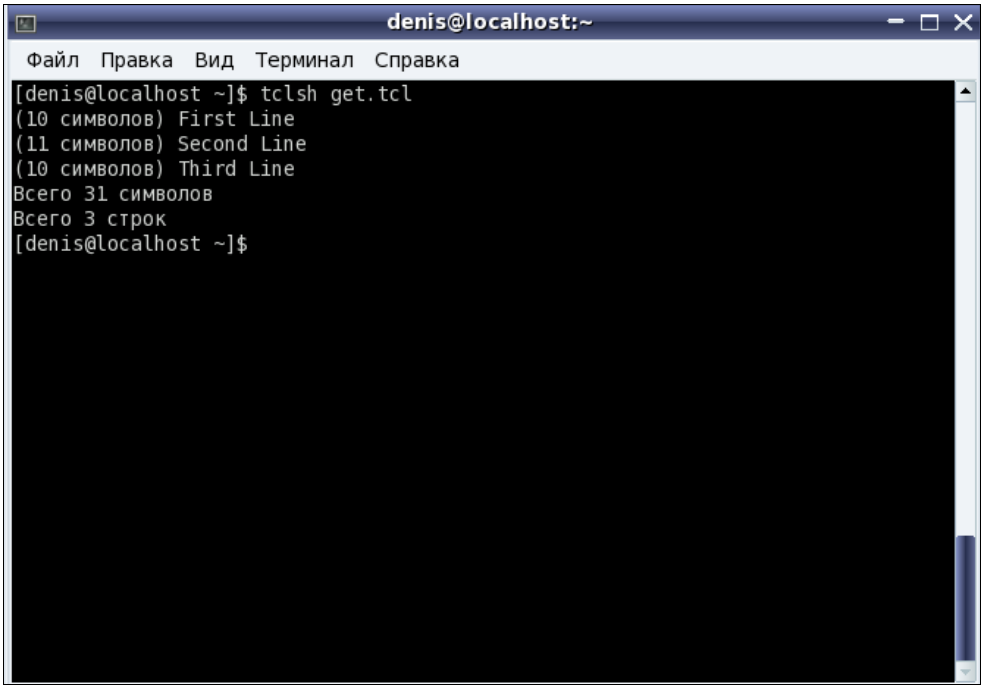


Рис. 25.1. Программа get.tcl

Команду read можно использовать как для чтения текстовых файлов, так и двоичных. Первый параметр — идентификатор канала, возвращаемый командой open. Второй — количество символов, которое должна прочитать read. Команда read может прочитать меньше символов, чем указано — если конец файла (EOF) будет достигнут раньше. Если количество символов не указано, read будет читать весь файл.

Рассмотрим два примера:

```
set fId [open test.txt r];
set input [read $fId];
puts "Прочитано [string length $input] символов";
close $fId;
```

Мы прочитали весь файл в одну строку input, а затем сообщили, сколько символов в этой строке. Теперь будем читать файл блоками по 512 символов:

```
set fId [open test.txt r];
while {![eof $fId]} {
set input [read $fId 512]
puts "Прочитано [string length $input] символов: $input"
}
close $fId;
```

Нужно отметить, что функция read работает гораздо быстрее, чем gets, поэтому для чтения больших файлов предпочтительнее использовать read. При чтении больших файлов нужно учитывать размер блока: чем он больше, тем выше скорость чтения файла. Максимальный размер блока для read зависит от файловой системы: не стоит превышать максимальный размер блока файловой системы. В большинстве случаев вполне хватит размера блока 1024 или 2048 символов.

Обычно возможностей команд gets и read вполне достаточно, поэтому изучение команды scan пусть будет вашим домашним заданием.

25.3. Запись файлов

Запись файлов осуществляется с помощью хорошо знакомой команды puts. Напомню формат ее вызова:

```
puts [-nonewline] [ID_канала] строка
```

По умолчанию puts выводит строку на стандартный вывод. Если же указан идентификатор канала, вывод будет осуществлен в файл, которому соответствует указанный идентификатор. Параметр -nonewline запрещает после строки выводить символ новой строки.

А вот здесь начинается самое интересное. При записи двоичных файлов или когда строка уже содержит символ новой строки, нужно указывать параметр -nonewline, чтобы puts не занималась самодеятельностью и чтобы после записи в файл вы обнаружили там то, что ожидали.

При записи текстовых файлов особо морочить себе голову с символом новой строки не стоит, хотя вы должны знать, что в разных операционных системах для обозначения конца строки используются следующие символы:

- `\n` — новая строка (UNIX, Linux, MacOS X);
- `\r` — перенос каретки (версии MacOS до OS X);
- `\r\n` — перенос каретки и новая строка (Windows).

Нужный символ новой строки будет добавлен автоматически, в зависимости от операционной системы, под управлением которой запущен интерпретатор TCL (поскольку в книге рассматривается программирование в Linux, то о версиях TCL для других ОС я предпочел умолчать). Если же вам хочется управлять тем, что `puts` записывает в файл, вы можете:

- использовать параметр `-nonewline` и вручную дописывать нужные символы (`puts "Hello\r"`);
- использовать команду настройки канала `fconfigure`. Данная команда не рассматривается в книге, но о ней вы можете прочитать по адресу: **http://www.tcl.tk/man/tcl8.4/TclCmd/fconfigure.htm**.

При использовании команды `puts` не забывайте использовать команду `format` *(см. главу 24)* для форматирования вывода.

После осуществления записи в файл желательно сбросить содержимое буферов ввода/вывода на диск, дабы убедиться, что данные действительно записаны в файл. Для этого используется команда `fflush`:

```
fflush ID_канала
```

25.4. Произвольный доступ к файлу

В других языках программирования мы можем получить произвольный доступ к файлу: мы можем переместить указатель позиции файла в любое место файла и осуществить операцию чтения или записи. Такая же возможность есть и в TCL. Для организации произвольного доступа к файлу используются команды `tell` и `seek`:

```
tell ID_канала
seek ID_канала смещение [отсчет]
```

Первая команда возвращает текущую позицию указателя файла. Вторая — перемещает указатель позиции файла. Смещение задает количество символов, на которое будет сдвинута текущая позиция файла от параметра `отсчет`. Параметр `отсчет` может принимать значения:

- `start` — начало файла;
- `end` — конец файла;
- `current` — смещение будет измеряться от текущей позиции файла.

Язык TCL редко используется в качестве языка для системного программирования, поэтому вряд ли вы будете использовать произвольный доступ к файлу — не вижу смысла останавливаться на нем подробно.

ГЛАВА 26

Понятие о виджетах

26.1. Tk-программирование

Наконец-то мы добрались до самого "вкусного", а именно до создания графического интерфейса для нашей TCL-программы. В *главе 23* мы написали простейшее графическое приложение, с вашего позволения я напомню вам его код, дабы вы лишний раз не листали книгу — код этого простого приложения нам сейчас пригодится (листинг 26.1).

Листинг 26.1. Простое приложение с графическим интерфейсом

```
#!/usr/bin/wish
label .l -width [string length "My First Tk-application"] -text "My First
Tk-application";
button .b -text "Exit" -command exit;
pack .l -padx 40 -pady 10
pack .b -padx 40 -pady 10
```

После того как вы познакомились с синтаксисом TCL, многое стало понятнее. Основное отличие приложения с графическим интерфейсом — другой интерпретатор. Вместо привычного `tclsh` нужно использовать интерпретатор `wish`.

Итак, если вам нужно написать обычную программу без графического интерфейса, добавьте в ее начало строку:

```
#!/usr/bin/tclsh
```

Если нужен графический интерфейс:

```
#!/usr/bin/wish
```

Конечно, в случаях, когда вы не планируете делать файл TCL-программы исполняемым, можно ничего не добавлять в начало файла. Тогда запустить программу можно так:

```
tclsh <имя файла>
```

или

```
wish <имя файла>
```

Вернемся к нашему простейшему приложению, отображающему окно с текстовым сообщением и кнопкой **Exit**. Команды `label` и `button` формируют соответственно текстовую надпись и кнопку, а команды `pack` управляют расположением виджетов в окне программы.

> ***Примечание***
>
> Виджет — это элемент графического интерфейса: окно, список, кнопка, полоска прокрутки и т. д. Виджеты обеспечивают функциональность графического интерфейса.

Наше простейшее приложение отображает два виджета: текстовую надпись и кнопку. Библиотека Tk предоставляет 45 команд, создающих различные виджеты.

Tk-программы обрабатывают различные события. События возникают вследствие действий пользователя, например пользователь нажал кнопку, произошло событие щелчка мышью на кнопке (в других языках программирования оно называется `onClick`). В программе есть код, который будет запущен для обработки данного события. В нашем случае для обработки нажатия кнопки **Exit** будет выполнена команда `exit`, закрывающая окно. Пользователь передвинул окно, произошло другое событие. Ваша программа не обязательно должна реагировать на все мыслимые события, достаточно только запрограммированных событий. Например, в случае с нашим простейшим приложением программа могла бы реагировать на перемещение окна, на изменение размера окна, на щелчок на области окна, на двойной щелчок на области окна и т. д. Но я запрограммировал реакцию только на одно событие — нажатие кнопки. Вместо стандартной TCL-команды `exit` вы можете указать свою процедуру, которая будет заниматься обработкой события. В этой части книги вы найдете пример реализации реакции на событие.

События — это тема для отдельного разговора, и мы о них еще поговорим в этой книге. А пока рассмотрим две занимательные таблицы. В табл. 26.1 приведены команды, "рисующие" основные элементы окна программы — кнопки, меню, текстовые надписи и т. д. А в табл. 26.2 приведены стандартные параметры виджетов. У каждого виджета свой набор параметров, но все Tk-виджеты имеют параметры, описанные в табл. 26.2. Стандартных опций очень много, но в табл. 26.2 приведены только те, которые вы будете использовать на практике. Некоторые бесполезные опции не включены в состав таблицы.

Таблица 26.1. *Основные виджеты*

Виджет	Описание
`button`	Создает обычную кнопку
`canvas`	Создает канву, на которой вы можете рисовать графические примитивы
`checkbutton`	Создает независимый переключатель (флажок)
`entry`	Редактируемое однострочное поле для ввода

Таблица 26.1 *(окончание)*

Виджет	Описание
`frame`	Контейнер, позволяющий размещать другие виджеты
`label`	Нередактируемое (только чтение) многострочное текстовое поле
`labelframe`	Создает фрейм, обладающий свойствами текстовой надписи (`label`)
`listbox`	Список с полосками прокрутки
`menu`	Меню
`menubutton`	Элемент меню
`message`	Многострочная текстовая надпись (только чтение)
`panedwindow`	Контейнер для отображения виджетов (панель)
`radiobutton`	Зависимый переключатель
`scale`	Шкала выбора значения параметра (такую можно использовать, например, для установки уровня громкости)
`scrollbar`	Полоска прокрутки
`spinbox`	Поле с кнопками уменьшения и увеличения значения
`text`	Поле для редактирования текста
`toplevel`	Контейнер (фрейм), который становится новым окном верхнего уровня

Таблица 26.2. *Стандартные параметры виджетов*

Опция	Описание
`-activebackground`	Устанавливает цвет фона активного элемента (активным элемент становится, когда он выбран с помощью клавиш управления курсором или же когда на элементе нажата кнопка мыши)
`-activeborderwidth`	Устанавливает ширину (в пикселах) рамки активного элемента
`-activeforeground`	Цвет переднего плана активного элемента
`-anchor`	Устанавливает позицию текста в виджете (допустимы значения `n`, `ne`, `e`, `se`, `s`, `sw`, `w`, `nw` и `center`)
`-background`	Цвет фона
`-bd`	Псевдоним для `-borderwidth`
`-bg`	Псевдоним для `-background`
`-bitmap`	Позволяет указать BMP-файл для отображения в виджете
`-borderwidth`	Ширина рамки
`-compound`	Задает расположение BMP-изображения относительно текста (допустимые значения: `bottom`, `top`, `left`, `right`, `center`)
`-cursor`	Позволяет изменить тип курсора, когда указатель мышки находится над виджетом

Таблица 26.2 *(окончание)*

Опция	Описание
`-disabledforeground`	Устанавливает цвет фона для отключенных виджетов
`-exportselection`	Определяет, должен или нет выделенный текст также быть X-выделением
`-fg`	Псевдоним для `-foreground`
`-font`	Определяет шрифт текста виджета
`-foreground`	Устанавливает цвет переднего плана для виджета
`-highlightbackground`	Устанавливает цвет фона подсвеченного региона виджета, который получил фокус ввода
`-highlightcolor`	Устанавливает цвет подсвеченного региона виджета, который получил фокус ввода
`-highlightthickness`	Устанавливает ширину подсвеченного региона виджета, получившего фокус ввода
`-image`	Устанавливает изображение, перезаписывая опции `-bitmap` и `-text`
`-justify`	Выравнивание для многострочного текстового поля. Может быть: `left`, `right` или `center`
`-padx`	Отступ слева и справа от сторон виджета
`-pady`	Отступ сверху и снизу от сторон виджета
`-relief`	3D-эффект виджета, может быть `flat`, `groove`, `raised`, `ridge`, `sunken` или `solid`
`-selectforeground`	Устанавливает цвет переднего плана для выбранных элементов
`-text`	Текст виджета
`-wraplength`	Устанавливает максимальную длину строки: если она превышена, текст будет перенесен

26.2. Компоненты Tk-приложения и имена виджетов

Еще раз запустите тестовое приложение и внимательно посмотрите на него. У нашего окна есть кнопки управления окном (минимизация, изменение размера, закрытие окна), есть заголовок окна, но все это не является Tk-виджетами. Эти элементы управления окном "рисует" менеджер окон, поэтому заголовок и кнопки управления окном в разных операционных системах будут выглядеть по-разному. Если есть возможность, попробуйте запустить тестовое приложение в Windows, MacOS и Linux — посмотрите, как будет изменяться внешний вид окна.

Примечание

В Linux TCL/Tk обычно установлен по умолчанию, а релизы для Windows и MacOS можно найти на странице **http://www.tcl.tk/software/tcltk/8.0.html**.

Если же лень устанавливать TCL/Tk в Windows, посмотрите хотя бы, как будет выглядеть ваше приложение в разных графических средах — GNOME и KDE. Заголовок и кнопки управления будут отличаться.

Внешний вид Tk-виджетов тоже зависит от операционной системы и оконного менеджера, но вы хотя бы можете повлиять на то, как виджет будет выглядеть: для настройки внешнего вида используются параметры виджета.

Теперь поговорим об именах виджетов. В нашем случае текстовая надпись называется `.l`, а кнопка — `.b`:

```
label .l -width [string length "My First Tk-application"] -text
"My First Tk-application";
button .b -text "Exit" -command exit;
```

Почему мы выбрали такие странные имена? Почему нельзя было назвать виджеты более естественно, например `button1` или `label1`?

Виджеты организуют иерархию, во главе которой находится главное окно приложения, которое обозначается как "." (точка). Чтобы показать, что кнопка или надпись относятся к главному окну приложения, мы указали точку в начале имени виджета. Что будет после точки — зависит от вашей фантазии, например кнопку можно было бы назвать `.button1`:

```
button .button1 -text "Exit" -command exit;
```

Однако о точке в начале забывать не стоит (иначе Tk просто не сможет определить, кому принадлежит виджет)! Если у нашей надписи появятся дочерние виджеты, например маленькие надписи, они должны называться так: `.l.small1` и `.l.small2`. Имя после последней точки может быть любым, но `.l` указывает, что новые надписи принадлежат родительской надписи `.l`, которая, в свою очередь, принадлежит главному окну приложения (`.`).

Имя виджета должно начинаться со строчной латинской буквы (нижний регистр), но для лучшей читабельности желательно, чтобы все буквы были в нижнем регистре. Имена виджетов могут содержать цифры. Имя не может начинаться с прописной буквы, поскольку такие имена относятся к ресурсам X.Org и во избежание конфликта имен виджеты принято называть со строчной буквы.

ГЛАВА 27

Основные элементы графического интерфейса

27.1. Команда *pack*

Мало просто перечислить виджеты в TCL-программе. Нужно еще зарегистрировать их расположение в окне программы, а это можно сделать с помощью менеджеров геометрии. В TCL используются менеджеры геометрии `pack`, `grid` и `place`. Менеджер `grid` будет рассмотрен в следующей главе, а в этой главе мы будем рассматривать менеджер `pack`, поскольку он является более популярным, чем `place`. Свою популярность он заслужил еще тем, что был самым первым менеджером геометрии, используемым в Tk. Поэтому, если вы используете `pack`, ваши программы будут одинаково отображаться на всех системах — даже на самых старых.

Важно понимать, как работает `pack`, иначе расположение элементов графического интерфейса станет для вас неприятным сюрпризом — вы увидите что угодно, кроме того, что ожидаете.

Менеджер `pack` размещает виджеты согласно упорядоченному списку упаковки, который формируется для каждого окна. С помощью опций `-in` (в), `-before` (перед) и `-after` (после) вы можете добавить новую запись в список упаковки. Другими словами, с помощью этих параметров задается место каждого виджета. Если эти опции не использовались, каждый новый виджет добавляется в конец списка упаковки свого предка (например, окна, в котором размещен виджет).

Попробуйте в нашем простом приложении first.tcl изменить последнюю строку так:

```
pack .b -padx 40 -pady 10 -before .l
```

Параметр `-before .l` означает, что кнопка должна быть размещена перед текстовой надписью. Результат изображен на рис. 27.1.

Последовательно просматривая список упаковки, менеджер `pack` размещает виджеты в окне. При обработке каждого виджета внутри окна есть прямоугольная незаполненная область, называемая полостью (cavity). Для первого виджета полость совпадает со всей поверхностью окна.

Рис. 27.1. Кнопка отображена до текстовой надписи

Для каждого виджета выполняются следующие действия:

- менеджер размещает прямоугольную область для очередного виджета у стороны полости, которая задана опцией `-side`. Если значение опции `-side` равно `top` или `bottom`, то ширина области равна ширине полости, а высота равна требуемой для размещения виджета высоте плюс внешние и внутренние поля, заданные опциями `-ipady` и `-pady`. Если опция `-side` равна `left` или `right`, высота области равна высоте полости, а ширина определяется размером виджета плюс полями, заданными опциями `-ipadx` и `-ipad`. Область может быть в одном или обоих направлениях с помощью опции `-expand`;
- менеджер определяет размеры виджета. Обычно они равны размерам, необходимым для виджета плюс внутренние поля, умноженные на 2. Однако размеры могут быть расширены до размеров области (минус удвоенные внешние поля) с помощью опции `-fill`;
- виджет размещается в отведенном ему пространстве. Если виджет меньше свободного пространства, учитывается значение опции `-anchor`;
- после размещения виджета выделенная ему область вычитается из полости. Полость для следующего виджета становится меньше, но тоже имеет прямоугольную форму. Если свободного места меньше, чем нужно для размещения виджета, то виджет получает столько пространства, сколько осталось. Если места не осталось, виджет (и все последующие виджеты в списке упаковки) не отображается на экране. Если же увеличить размеры окна, виджет появится на экране.

Для лучшего понимания алгоритма работы `pack` нужно рассмотреть опции этой команды (табл. 27.1). В табл. 27.1 приведены не все опции команды `pack`, с остальными вы познакомитесь в справочной системе (`man pack`).

Таблица 27.1. *Опции команды* `pack`

Опция	Описание
`-after мастер`	Виджет будет расположен после мастер-виджета
`-ahchor якорь`	Задает позицию якоря, по умолчанию используется значение c (центр)
`-before мастер`	Виджет будет расположен до мастер-виджета
`-expand 1|0`	Определяет, будут ли перечисленные виджеты расширяться (1) при наличии свободного места в мастер-окне или нет (0)

Таблица 27.1 *(окончание)*

Опция	Описание
`-fill стиль`	Если в мастер-окне для виджета есть свободное пространство, вы можете задать стиль растяжения виджета: • `none` — растяжения нет, предоставляется требуемый режим (значение по умолчанию); • `x` — горизонтальное растяжение; • `y` — вертикальное растяжение; • `both` — расширяет виджет в обоих направлениях
`-in мастер`	Помещает виджет в конец списка размещаемых виджетов для окна `мастер`
`-ipadx значение`	Указывает размер горизонтальных внутренних полей, которые должны быть с обеих сторон виджета. Размер можно указывать в пикселах, например `1` (1 пиксел), или в сантиметрах, например `1c` (0,1 см). Значение по умолчанию — 0
`-ipady значение`	Размер вертикальных внутренних полей. Значение по умолчанию — 0
`-padx значение`	Размер горизонтальных внешних полей. Значение по умолчанию — 0
`-pady значение`	Размер вертикальных внешних полей. Значение по умолчанию — 0
`-side сторона`	Определяет, к какой стороне мастер-окна будут привязаны виджеты: • `left` — к левой; • `right` — к правой; • `top` — к верхней; • `bottom` — к нижней. Значение по умолчанию — top
`forget виджет`	"Забыть" о виджете и удалить его из списка упаковки, виджет больше не будет отображаться на экране
`slaves окно`	Выводит список всех дочерних виджетов

27.2. Команда *button*

Команда `button` создает кнопку. Формат вызова этой команды следующий:

```
button имя опции
```

С именем виджета все понятно, а опции команды `button` приведены в табл. 27.2.

Таблица 27.2. *Опции команды* `button`

Опция	Описание
`-command`	Задает команду, которая будет выполнена при нажатии кнопки. Командой может быть как встроенная TCL-команда, так и пользовательская процедура

***Таблица 27.2** (окончание)*

Опция	Описание
`-default`	Определяет состояние по умолчанию: • `normal` — обычное состояние; • `active` — активное состояние; • `disabled` — кнопка неактивна
`-height`	Высота кнопки
`-state`	Состояние кнопки: • `normal` — обычное состояние; • `active` — активное состояние; • `disabled` — кнопка неактивна
`-width`	Ширина кнопки
`invoke`	Принудительно выполняет команду, указанную параметром `-command`. Полезно, когда нужно симулировать нажатие кнопки пользователем
`configure`	Позволяет изменять параметры уже созданной кнопки
`flash`	Создает эффект мерцания кнопки

Рассмотрим небольшой пример работы с кнопками. Создадим окно с двумя кнопками — **Activate** и **Exit**. По умолчанию кнопка **Exit** будет неактивной, при нажатии кнопки **Activate** произойдет активация кнопки **Exit** и тогда можно будет ею воспользоваться для выхода из программы (листинг 27.1).

Листинг 27.1. Сценарий buttons.tcl

```
set bExit [button .e -width 20 -text "Exit" -state disabled \
-command exit]
set bActivate [button .a -width 20 -text "Activate" \
-command {$bExit configure -state normal}]

pack $bActivate -pady {0 20}
pack $bExit
```

Проанализируем код этой простой команды. Первая команда создает кнопку **Exit**, которая по умолчанию неактивна (`-state disabled`). Вторая кнопка — самая обычная кнопка с надписью **Activate**. Посмотрите на команду кнопки **Activate**:

```
$bExit configure -state normal
```

`$bExit` — это имя переменной виджета, представляющего собой кнопку **Exit**. Затем мы используем опцию `configure` для изменения состояния кнопки с неактивного на нормальное. Мы не можем использовать команду `button .e -state normal`, поскольку ее вызов приведет к добавлению еще одной кнопки с именем `.e`, что недо-

пустимо. Поэтому для изменения параметров уже созданной кнопки нужно использовать команду `configure`.

Процесс разработки, запуска приложения, а также само приложение изображены на рис. 27.2.

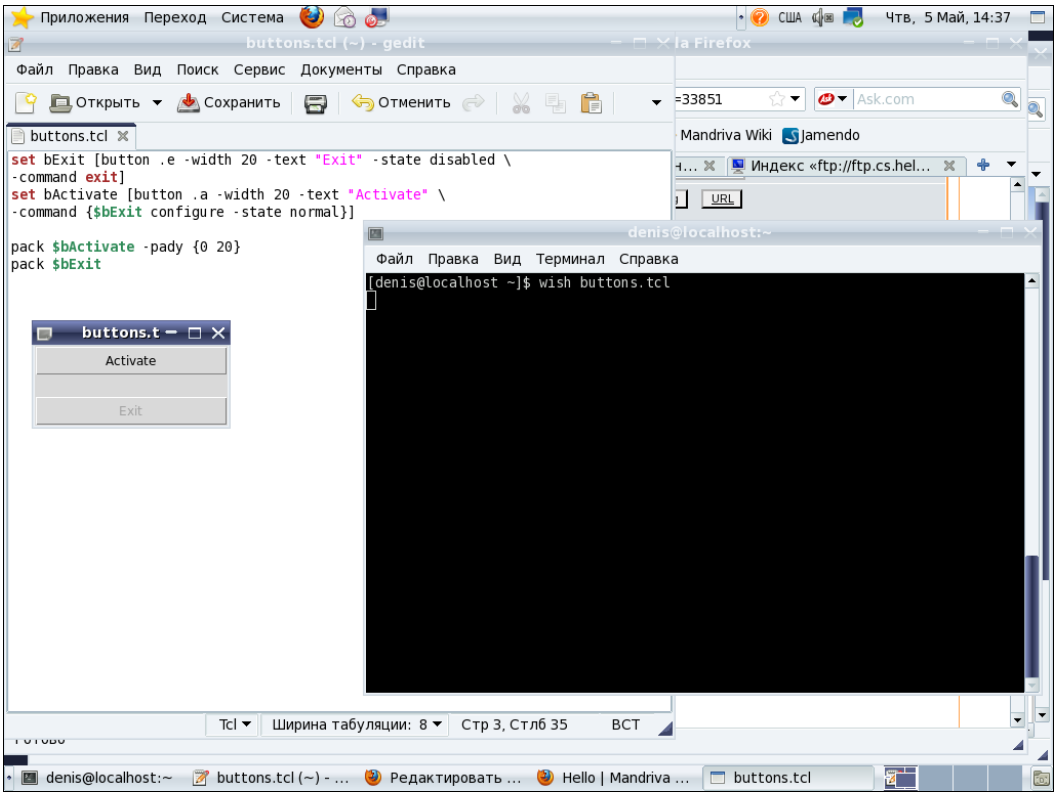


Рис. 27.2. Разработка и запуск приложения buttons.tcl

Рассмотрим еще один пример. Кнопка **Exit** уже не будет неактивной, а вместо кнопки **Activate** будет кнопка **Flash**, заставляющая мерцать кнопку **Exit** (листинг 27.2). Изменения в нем минимальны, поэтому вы сами разберетесь, что к чему.

Листинг 27.2. flash_buttons.tcl

```
set bExit [button .e -width 20 -text "Exit" \
-command exit]
set bFlash [button .a -width 20 -text "Flash" \
-command {$bExit flash}]

pack $bFlash -pady {0 20}
pack $bExit
```

27.3. Команда *checkbutton*

Независимые переключатели (флажки) создаются с помощью команды `checkbutton`:

```
checkbutton имя опции
```

Как и у команды `button`, у команды `checkbutton` есть свой набор опций (табл. 27.3).

Таблица 27.3. *Опции команды* `checkbutton`

Опция	Описание
`-command`	Задает команду, которая будет выполнена при нажатии переключателя. Командой может быть как встроенная TCL-команда, так и пользовательская процедура
`-height`	Высота переключателя
`-indicatoron`	Позволяет определить, включен или выключен переключатель
`-offvalue`	Определяет значение, передаваемое, если переключатель выключен. По умолчанию передается значение 0
`-onvalue`	Определяет значение, передаваемое, если переключатель включен. По умолчанию передается значение 1
`-selectcolor`	Определяет цвет фона переключателя, когда переключатель включен
`-state`	Статус переключателя: `normal`, `active`, `disabled`
`-variable`	Определяет имя переменной, ассоциированной с переключателем. По этой переменной можно определить, включен переключатель или выключен
`-width`	Ширина переключателя
`cget`	Возвращает значение указанной опции
`configure`	Используется для установки определенной опции
`flash`	Эффект мерцания переключателя
`invoke`	Принудительное выполнение команды, заданной аргументом `-command`
`select`	Выбирает переключатель и присваивает переменной, указанной с помощью параметра `-variable`, значение, заданное параметром `-onvalue`
`deselect`	Выключает переключатель и присваивает переменной, указанной с помощью параметра `-variable`, значение, заданное параметром `-offvalue`
`toggle`	Изменяет состояние переключателя на противоположное. Если он был выключен, включает его и наоборот

Теперь напишем программу, демонстрирующую работу с переключателями. Первым делом опишем три переключателя:

```
set ckFirst [checkbutton .ckfirst -text "Первый переключатель" \
-command {GetState .ckfirst}]
set ckSecond [checkbutton .cksecond -text "Второй переключатель" \
```

```
-command {GetState .cksecond}]
set ckThird [checkbutton .ckthird -text "Третий переключатель" \
-offvalue "OFF" -onvalue "ON" -command {GetState .ckthird}]
```

Мы определили три переменные: `ckFirst`, `ckSecond` и `ckThird`, имена соответствующих им переключателей — `.ckfirst`, `.cksecond`, `.ckthird`. Команда для всех переключателей одна: будет запущена процедура `GetState`, которой передается имя переключателя. Процедура `GetState` выводит в терминал значение переменной, связанной с переключателем (именно поэтому демонстрационную программу нужно запускать в терминале). По умолчанию, если переключатель выбран, передается значение 1, а если нет — 0. С помощью параметров `-onvalue` и `-offvalue` мы можем изменять передаваемые значения, что мы и сделали для третьего переключателя. Когда он выбран, будет передаваться значение `"ON"`, а когда нет — `"OFF"`. Процедура `GetState` выглядит так:

```
proc GetState {ckbuttons} {
foreach ckbutton $ckbuttons {
       set var [$ckbutton cget -variable]
       global $var
       set val [set $var]
       puts "$var = $val" }
}
```

Получение значения переключателя происходит с помощью команды `$ckbutton cget -variable`.

После определения всех трех переключателей нужно объявить список переключателей `ckbuttons`, который мы будем использовать для совершения операций над всеми переключателями сразу:

```
set ckbuttons [list $ckFirst $ckSecond $ckThird]
```

Теперь объявим пять кнопок:

- **Команда Toggle** — выполняет команду `toggle` для всех переключателей, в результате чего состояние каждого переключателя будет заменено на противоположное;
- **Команда Clear** — сбрасывает все переключатели;
- **Команда Set** — включает все переключатели;
- **Показать** — показывает состояние всех переключателей;
- **Выход** — завершает работу приложения.

Вот все эти кнопки:

```
set bToggleAll [button .toggleall -width 12 -text "Команда Toggle" \
-command {ButtonCommand toggle $ckbuttons}]
set bClearAll [button .clearall -width 12 -text "Команда Clear" \
-command {ButtonCommand clear $ckbuttons}]
set bSetAll [button .setall -width 12 -text "Команда Set" \
```

```
-command {ButtonCommand set $ckbuttons}]
set bShowAll [button .showall -width 12 -text "Показать" \
-command {GetState $ckbuttons}]
set bExit [button .exitb -width 12 -text "Выход" \
-command exit]
```

Для первых трех кнопок в качестве действий вызывается процедура `ButtonCommand`, которой передается два параметра — действие, которое нужно совершить над кнопкой, и список `ckbuttons` (чтобы действие было выполнено над всеми переключателями). Четвертая кнопка вызывает процедуру `GetState` для списка `ckbuttons`, т. е. для всех переключателей сразу. Последняя кнопка — классическая кнопка завершения работы программы.

Процедура `ButtonCommand` выглядит так:

```
proc ButtonCommand {action ckbuttons} {
foreach ckbutton $ckbuttons {
switch $action \
       toggle { $ckbutton toggle } \
       clear { $ckbutton deselect } \
       set { $ckbutton select } }
}
```

Для определения нужного действия используется команда `switch`, которая ранее не была рассмотрена в этой книге. Однако разобраться с ней, думаю, особого труда не составит, поскольку подобные конструкции есть в любом языке программирования.

Полный код программы приведен в листинге 27.3.

Листинг 27.3. Программа check.tcl

```
proc ButtonCommand {action ckbuttons} {
foreach ckbutton $ckbuttons {
switch $action \
toggle { $ckbutton toggle } \
clear { $ckbutton deselect } \
set { $ckbutton select } }
}

proc GetState {ckbuttons} {
foreach ckbutton $ckbuttons {
set var [$ckbutton cget -variable]
global $var
set val [set $var]
puts "$var = $val" }
}

set ckFirst [checkbutton .ckfirst -text "Первый переключатель" \
-command {GetState .ckfirst}]
```

```
set ckSecond [checkbutton .cksecond -text "Второй переключатель" \
-command {GetState .cksecond}]
set ckThird [checkbutton .ckthird -text "Третий переключатель" \
-offvalue "OFF" -onvalue "ON" -command {GetState .ckthird}]

set ckbuttons [list $ckFirst $ckSecond $ckThird]

set bToggleAll [button .toggleall -width 12 -text "Команда Toggle" \
-command {ButtonCommand toggle $ckbuttons}]
set bClearAll [button .clearall -width 12 -text "Команда Clear" \
-command {ButtonCommand clear $ckbuttons}]
set bSetAll [button .setall -width 12 -text "Команда Set" \
-command {ButtonCommand set $ckbuttons}]
set bShowAll [button .showall -width 12 -text "Показать" \
-command {GetState $ckbuttons}]
set bExit [button .exitb -width 12 -text "Выход" \
-command exit]

pack $ckFirst $ckSecond $ckThird -anchor w
pack $bToggleAll -pady 5 -padx 5 -side left
pack $bClearAll $bSetAll $bShowAll $bExit -pady 5 -padx {0 5} -side left
```

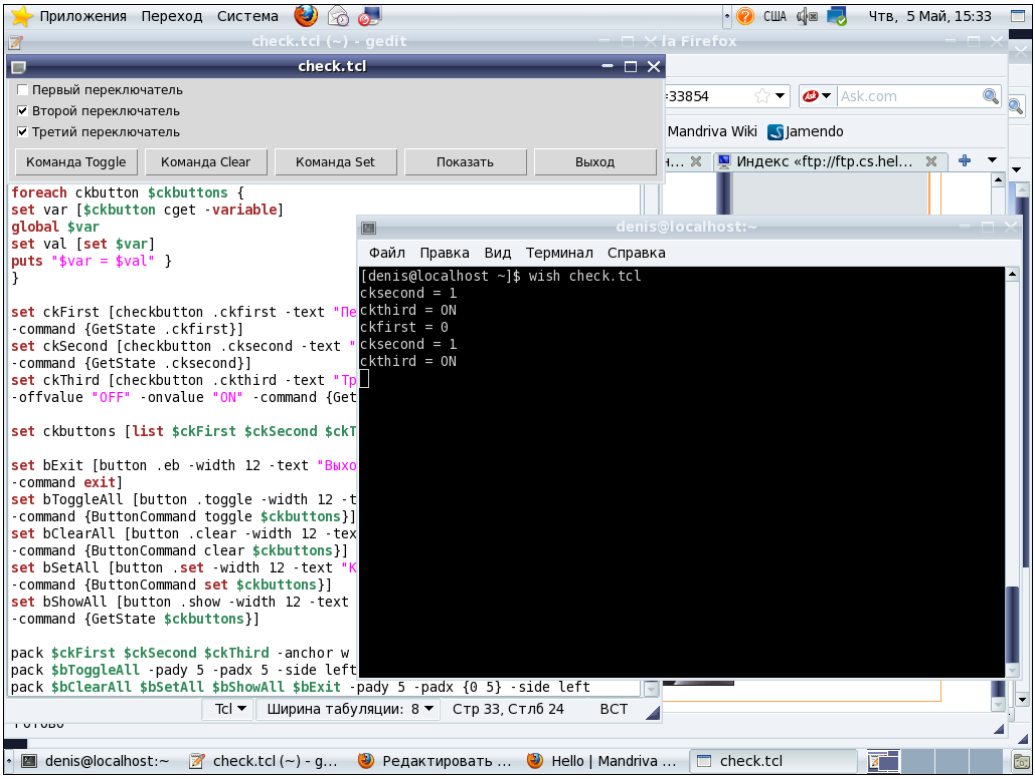


Рис. 27.3. Программа check.tcl в действии

На рис. 27.3 программа изображена в действии. Посмотрите на вывод программы на терминале. Сначала я вручную включил переключатели 2 и 3, поэтому программа вывела строки:

```
cksecond = 1
ckthird = ON
```

Затем я нажал кнопку **Показать** для вывода значений всех трех переключателей:

```
ckfirst = 0
cksecond = 1
ckthird = ON
```

27.4. Зависимые переключатели

Зависимый переключатель отличается от независимого тем, что его значение зависит от значений других переключателей. Если у нас есть три независимых переключателя, то мы можем как угодно устанавливать значения каждого из них, установка значения одного из переключателей никак не отобразится на значениях других переключателей. А вот если у нас есть три зависимых переключателя, то активным может быть только один из них — остальные будут выключены. Зависимые переключатели удобно использовать, когда нужно предоставить пользователю возможность выбрать одно значение из нескольких возможностей.

Создать зависимый переключатель можно командой `radiobutton`. Набор опций у нее такой же, как у команды `checkbutton`. Рассмотрим применение команды `radiobutton` на практике. Мы создадим окно с тремя зависимыми переключателями, позволяющими выбрать цвет фона основного окна. Кнопка **Установить цвет фона** будет использоваться для установки цвета, а кнопка **Выход** — для выхода из приложения.

Сначала описываются зависимые переключатели:

```
set rRed [radiobutton .rred -text "Красный" -value red]
set rBlue [radiobutton .rblue -text "Синий" -value blue]
set rGreen [radiobutton .rgreen -text "Зеленый" -value green]
```

Затем описывается список `radioButtons`, используемый для управления всеми переключателями сразу:

```
set radioButtons [list $rRed $rGreen $rBlue]
```

После описываем текстовую надпись и наши кнопки управления приложением:

```
set labelColor [label .lcolor -text "Выберите цвет"]
set bSet [button .bset -width 15 -text "Установить цвет фона" \
-command {SetColor $selectedButton}]

set bExit [button .bexit -width 15 -text "Выход" -command exit]
```

Кнопка **Установить цвет фона** вызывает процедуру `setColor`, которой передается выбранный переключатель. Процедура `setColor` устанавливает цвет фона в зависи-

мости от выбранного переключателя. Потом мы программно устанавливаем переключатель по умолчанию и также программно "нажимаем" кнопку `bSet`, что приводит к установке красного цвета фона при запуске программы:

```
$rRed select
$bSet invoke
```

Полный код программы приведен в листинге 27.4. Программа radio.tcl изображена на рис. 27.4.

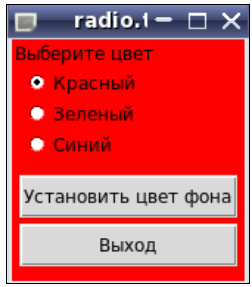


Рис. 27.4. Программа radio.tcl

Листинг 27.4. Программа radio.tcl

```
proc SetColor {newColor} {

. configure -background $newColor

global labelColor

$labelColor configure -background $newColor
global radioButtons
foreach button $radioButtons {
        $button configure -background $newColor \
        -activebackground $newColor \
        -highlightbackground $newColor }
}

set rRed [radiobutton .rred -text "Красный" -value red]
set rBlue [radiobutton .rblue -text "Синий" -value blue]
set rGreen [radiobutton .rgreen -text "Зеленый" -value green]

set radioButtons [list $rRed $rGreen $rBlue]

set labelColor [label .lcolor -text "Выберите цвет"]

set bSet [button .bset -width 15 -text "Установить цвет фона" \
-command {SetColor $selectedButton}]

set bExit [button .bexit -width 15 -text "Выход" -command exit]
```

```
$rRed select
$bSet invoke
pack $labelColor -anchor w
pack $rRed $rGreen $rBlue -pady 0 -padx {2 0} -anchor w
pack $bSet -pady {10 2} -padx 5
pack $bExit -pady {2 10} -padx 5
```

27.5. Создание меню

Меню создается командой `menu`. Сначала создается главное меню приложения:

```
set mainMenu [menu .mainmenu]
```

Затем нужно добавить в него подменю, например **Файл**:

```
set mFile [menu $mainMenu.mFile -tearoff 0]
$mainMenu add cascade -label "Файл" -menu $mFile
```

Далее в каждое подменю нужно добавить команды:

```
$mFile add command -label "Открыть..." -command {Cmd Open}
$mFile add command -label "Закрыть" -command {Cmd Close}
```

Параметр `add command` добавляет в меню новую команду с меткой `label`, при выборе пункта меню будет выполнена команда, указанная параметром `command`. Здесь `Cmd` — это обычная TCL-процедура, которой передается один параметр — название пункта меню.

Добавить разделитель в меню можно опциями `add separator`:

```
$mFile add separator
```

Вроде бы все понятно, напишем небольшую программу с меню **Файл** и **Правка** (листинг 27.5). Результат работы программы представлен на рис. 27.5.

Листинг 27.5. Программа menu.tcl

```
set mainMenu [menu .mainmenu]
. configure -menu $mainMenu
set mFile [menu $mainMenu.mFile -tearoff 0]
$mainMenu add cascade -label "Файл" -menu $mFile

$mFile add command -label "Создать..." -command {Cmd CreateFile}
$mFile add command -label "Открыть..." -command {Cmd Open}
$mFile add command -label "Сохранить" -command {Cmd Save}
$mFile add command -label "Сохранить как" -command {Cmd SaveAs}
$mFile add separator
$mFile add command -label "Печать" -command {Cmd Print}
$mFile add command -label "Параметры печати" -command {Cmd PrintSetup}
$mFile add separator
$mFile add command -label "Закрыть" -command {Cmd Close}
$mFile add command -label "Выход" -command exit
```

```
set mEdit [menu $mainMenu.mEdit -tearoff 0]
$mainMenu add cascade -label "Правка" -menu $mEdit
$mEdit add command -label "Копировать" -command {Cmd "Copy"}
$mEdit add command -label "Вырезать" -command {Cmd "Cut"}
$mEdit add command -label "Вставить" -command {Cmd "Paste"}
$mEdit add separator
$mEdit add command -label "Найти" -command {Cmd "Search"}
$mEdit add command -label "Заменить" -command {Cmd "Replace"}

proc Cmd {cmd} {
tk_messageBox -icon info -type ok -message $cmd
}
```

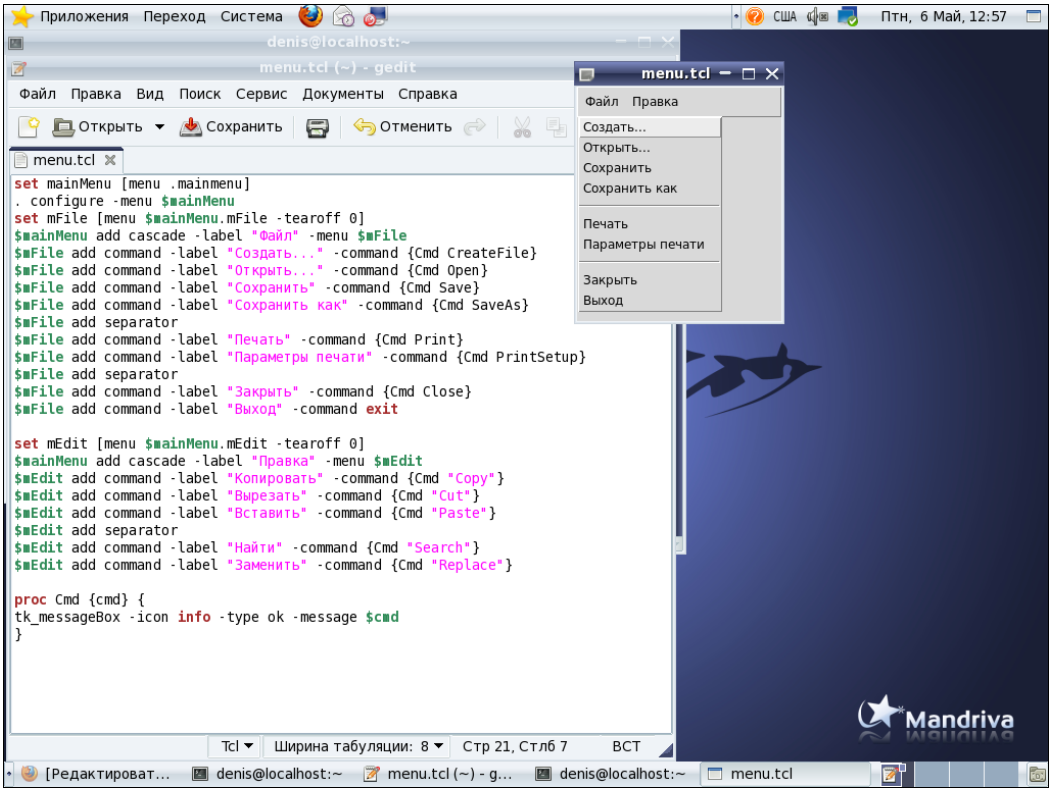


Рис. 27.5. Программа menu.tcl

Обратите внимание на команды меню. Команда **Выход** вызывает процедуру `exit`, завершающую работу программы. Все остальные вызывают процедуру `Cmd`, которая вместо привычной нам команды `puts` вызывает команду `tk_messageBox`, отображающую диалоговое окно с сообщением, указанным в параметре `-message`. В нашем случае будет показано название выбранной команды.

27.6. Поля ввода

Библиотека Tk предлагает два поля ввода: `entry` и `spinbox`. Первое — это классическое текстовое поле ввода, подходит для ввода однострочного текста и цифр. Второе — это поле со счетчиком, подходит для ввода числовых значений.

У виджетов `entry` и `spinbox` примерно одинаковый набор атрибутов, поэтому мы рассмотрим их в одной большой табл. 27.4. В колонке "Виджет" этой таблицы будет указано, к какому виджету относится тот или иной атрибут.

***Таблица 27.4.** Атрибуты виджетов `entry` и `spinbox`*

Виджет	Опция	Описание
spinbox	-buttonbackground	Цвет фона для кнопок поля со счетчиком
spinbox	-buttoncursor	Курсор, отображаемый когда указатель мыши находится над кнопками `spinbox`'а
spinbox	-command	Команда, вызываемая, когда нажимаются кнопки поля со счетчиком
оба	-exportselection	Если `true`, выделение будет экспортироваться и станет X-выделением
оба	-format	Определяет формат строки. Для вещественных значений формат должен быть `%<pad>.<pad>f`
spinbox	-from	Задает начальное значение поля со счетчиком
оба	-insertbackground	Цвет фона области вставки
оба	-insertborderwidth	Ширина границы области вставки
оба	-insertofftime	Если не 0, курсор будет мигать и это значение определяет продолжительность стадии `off` цикла мерцания в миллисекундах
оба	-insertontime	Если не 0, курсор будет мигать и это значение определяет продолжительность стадии `on` цикла мерцания в миллисекундах
оба	-insertwidth	Определяет ширину курсора вставки (в пикселах)
оба	-invcmd	Определяет команду, которая будет запущена, если `-validatecommand` вернет 0
spinbox	-increment	Устанавливает шаг между значениями `-from` и `-to`. Предположим, что нужно создать поле, позволяющее вводить значения от 0 до 20 с шагом 2. Параметр `-from` нужно установить в 0, параметр `-to` в 20, параметр `increment` в 2
spinbox	-readonlybackground	Определяет цвет виджета, когда редактирование запрещено (только чтение)
оба	-selectbackground	Цвет фона выделенного текста
оба	-selectforeground	Цвет переднего плана выделенного текста

Таблица 27.4 *(окончание)*

Виджет	Опция	Описание
entry	-show	Маскирует содержимое текстового поля определенным символом, например *. Подходит для создания полей ввода пароля
оба	-textvariable	Имя переменной, значение которой будет привязано к атрибуту -text. Обновление этой переменной сразу отобразится на атрибуте -text
spinbox	-to	Конечное значение для поля со счетчиком
оба	-validate	Определяет режим проверки ввода: • none — без проверки ввода (по умолчанию); • focus — команда проверки правильности ввода будет запущена, когда виджет получает и теряет фокус; • focusin — команда проверки правильности ввода будет запущена, когда виджет получает фокус; • focusout — команда проверки правильности ввода будет запущена, когда виджет теряет фокус; • key — проверка правильности ввода при редактировании виджета; • all — команда проверки правильности ввода будет запущена во всех перечисленных случаях
оба	-vcmd	Определяет команду проверки правильности ввода, которая будет запущена, если -validate не равно none. Синоним для -vcmd: -validatecommand
оба	-values	Определяет список правильных значений. Для spinbox эти значения имеют более высокий приоритет, чем значения, заданные -from, -to и -increment

В табл. 27.4 приведены не все опции виджетов entry и spinbox, некоторые редко используемые виджеты не описаны. Полный набор параметров представлен в справочной системе:

```
man entry
man spinbox
```

Особого внимания заслуживает процедура проверки правильности ввода, задаваемая параметром -vcmd. В качестве параметра этой функции можно указывать подстановки из табл. 27.5.

Таблица 27.5. *Подстановки для процедуры проверки правильности ввода*

Подстановка	Описание
%d	Тип проверки (0 — для удаления, 1 — для вставки, –1 — для фокуса)
%i	Индекс вставленного/удаленного символа
%P	Значение виджета после редактирования

Таблица 27.5 *(окончание)*

Подстановка	Описание
%s	Значение виджета до редактирования
%S	Вставленная/удаленная строка
%v	Режим проверки (none, all, focus и т. д.)
%W	Название виджета

Рассмотрим программу, выводящую два текстовых поля. Первое поле предназначено для ввода прописных букв, а второе — только для ввода цифр. Для проверки значений виджетов будут использованы две процедуры. Как только пользователь введет неправильный символ (например, строчной в первое поле или любую букву во второе поле), цвет фона виджета будет изменен на красный.

Листинг 27.6. Проверка правильности ввода

```
# Процедура проверки первого поля
proc Letter {text} {
        if {[regexp {[^A-ZA-Я]} $text]} {
                .a configure -bg red
                return 0
        } else {
                .a configure -bg white
                return 1
        }
}

# Процедура проверки второго поля
proc Digit {text} {
        if {[regexp {[^0-9]} $text]} {
                .d configure -bg red
                return 0
        } else {
                .d configure -bg white
                return 1
        }
}


label .la -text "Прописные буквы:"
entry .a -bg white -textvariable a \
         -invcmd bell -validate all -vcmd {Letter %P}

pack .la .a -fill x
```

```
label .ld -text "Цифры:"
entry .d -bg white -textvariable d \
       -invcmd bell -validate all -vcmd {Digit %P}

pack .ld .d -fill x
```

Программа entry.tcl изображена на рис. 27.6.

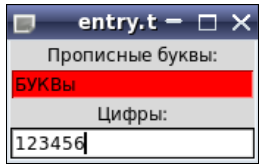


Рис. 27.6. Программа entry.tcl

27.7. Списки и домашнее задание

Список создается командой `listbox`. В этой главе мы напишем программу, загружающую элементы списка из текстового файла. В текстовом файле элементы находятся по одному в каждой строке. Код программы представлен в листинге 27.7.

Листинг 27.7. Программа list.tcl

```
# Процедура загрузки элементов списка
proc GetCars {carsfile} {
set fileId [open $carsfile r]
while {[gets $fileId line] > 0} {
lappend cars [string trim [lrange $line 0 end]]
}
close $fileId
return $cars
}

# Список, фон — белый
listbox .carslist -bg white

# Загружаем элементы списка
set cars [GetCars "cars.txt"]
foreach car $cars {
.carslist insert end $car
}

# Отображаем список и кнопку "Выход"
button .bexit -text "Выход" -command exit
grid .carslist -padx 5 -pady 5 -sticky nsew
grid .bexit -pady {0 5}
```

Данная программа (рис. 27.7) немного отличается от тех, которые мы создавали раньше. Вместо менеджера геометрии `pack` она использует менеджер `grid`, который будет рассмотрен в следующей главе.

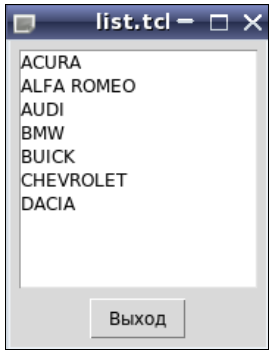


Рис. 27.7. Программа list.tcl

В качестве домашнего задания я предлагаю рассмотреть команды `listbox` (список), `scrollbar` (полоска прокрутки) и `text` (многострочное текстовое поле). Помочь в освоении этих виджетов вам помогут уже готовые примеры, которые уже установлены на вашем компьютере. Вы найдете их в каталоге /usr/share/tk8.6/demos (версия Tk может отличаться, поэтому название каталога может быть несколько другим). Каждый пример тщательно закомментирован, поэтому разобраться с ним, зная синтаксис TCL, не составит труда. А мы же займемся более сложными материями — привязкой событий и программированием окон (в следующей главе).

27.8. Программирование событий. Команда *bind*

Я обещал, что мы поговорим о программировании событий, или о привязке (в терминологии Tk) к событиям. Команда `bind` выполняет команду, если произошло указанное событие (табл. 27.6) для указанного виджета:

```
bind виджет событие команда
```

Другими словами, команда `bind` выполняет привязку события к команде. Поскольку одни и те же события (например, щелчок мыши или нажатие клавиши) могут произойти для разных элементов графического интерфейса, нам нужно уточнить, какой виджет мы имеем в виду.

Таблица 27.6. *События*

Событие	Описание
`Active`	Окно активировано, получен фокус
`ButtonPress`	Кнопка мыши нажата, событие `Button` является синонимом для `ButtonPress`. `ButtonPress` — это не щелчок мышью. Щелкнуть мышью — это нажать и быстро отпустить кнопку мыши. Событие `ButtonPress` — кнопка мыши нажата, но не отпущена

Таблица 27.6 *(окончание)*

Событие	Описание
ButtonRelease	Клавиша мыши отпущена
Configure	Окно перемещено или был изменен его размер
Deactivate	Фокус перешел к другому элементу
Destroy	Уничтожение элемента графического интерфейса (например, окна)
Enter	Событие относится к окну: указатель мыши вошел в пределы окна
Expose	Генерируется, когда окно должно быть перерисовано
FocusIn	Когда окно получило фокус ввода с клавиатуры
FocusOut	Окно "потеряло" фокус клавиатуры
KeyPress	Генерируется, когда нажата клавиша. Синоним для этого события — Key
KeyRelease	Генерируется, когда клавиша отпущена
Leave	Указатель мыши вышел за пределы окна
Map	Генерируется, когда окно появилось на экране (например, когда было свернуто, а потом восстановлено)
Motion	Генерируется при перемещении указателя мыши
MouseWheel	Генерируется при прокрутке с помощью колесика мыши
Unmap	Генерируется, когда окно стало невидимым на экране, например было минимизировано
Visibility	Происходит, когда видимость нашего окна изменяется, например другое окно появилось над нашим окном

ГЛАВА 28

Многооконный интерфейс

28.1. Менеджер геометрии *grid*

До этого мы создавали простые приложения, состоящие всего из одного окна. Но на практике все гораздо сложнее: редко программа состоит из одного окна, как минимум окон два-три. Прежде чем научиться создавать новые окна, мы рассмотрим менеджер геометрии `grid`, а затем уже перейдем к созданию многооконного интерфейса. Также в этой главе будут рассмотрены диалоги открытия и сохранения файла.

Менеджер геометрии `grid` размещает виджеты в виде сетки, что позволяет добиться более прогнозируемых результатов, чем с менеджером `pack`. При использовании `grid` вы можете применять одну из двух моделей размещения виджетов:

- относительное размещение — размеры каждого ряда и каждой колонки вычисляются в зависимости от содержимого каждой ячейки. Чем больше виджет, размещенный в ячейке, тем больше размер ячейки;
- абсолютное размещение — позиции каждого виджета указываются явно с использованием аргументов `-row` и `-column`.

28.1.1. Относительное размещение

Когда используется модель относительного позиционирования, каждый вызов команды `grid` создает новый ряд, а количество колонок определяется числом виджетов в этом ряду. Виджеты появляются в том порядке, в котором они были "отправлены" в ряд. Ширина колонки устанавливается по ширине самого большого виджета, а остальные, меньшие виджеты, выравниваются по центру в ячейке.

Аналогично, высота ряда определяется по самому высокому виджету. Остальные виджеты выравниваются по центру в своих ячейках. Рассмотрим программу, размещающую 12 надписей в таблице 3×4 (листинг 28.1).

Листинг 28.1. Относительное позиционирование с помощью `grid`

```
wm title . "Сетка"
for {set i 1} {$i <= 12} {incr i} {
```

```
        frame .fr$i -bg white -width .50i -height .50i
        label .l$i -bg blue -fg white -text $i
        grid propagate .fr$i false
        grid .l$i -in .fr$i
        if {![expr $i % 3]} {
        grid .fr[expr $i — 2] .fr[expr $i -1] .fr$i -padx 5 -pady 5
}
}
```

Данная глава посвящена операциям с окнами, поэтому программа начинается с изменения заголовка окна. Изменение заголовка окна, а также другие операции (сворачивание, изменение размера и позиции окна) с окном можно выполнить с помощью команды `wm`.

Далее в цикле `for` создается 12 фреймов с именами `.frN`, где `N` — номер фрейма, фон фреймов устанавливается белым, а размеры задаются опциями `-width` и `-height`. После создания фрейма создаются надписи с именами `.lN` (`N` — номер надписи), номер надписи соответствует номеру фрейма. Каждая надпись с помощью опции `-in` команды `grid` помещается в соответствующий ей фрейм:

```
grid .l$i -in .fr$i
```

Благодаря опции `-in` надписи становятся дочерними элементами фреймов.

Для каждого фрейма мы с помощью следующей команды запрещаем изменение размеров (чтобы все фреймы выглядели одинаково):

```
grid propagate .fr$i false
```

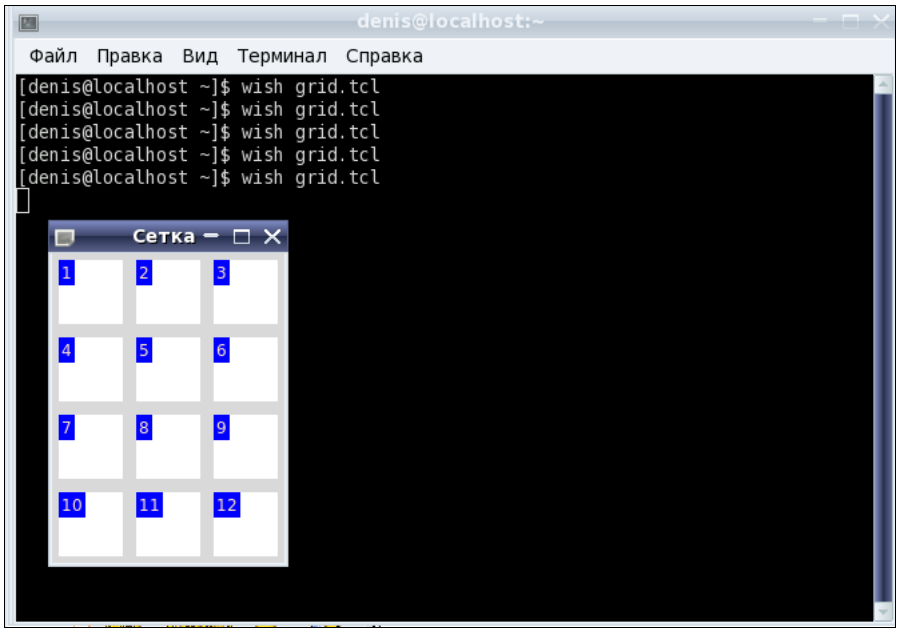


Рис. 28.1. Программа grid.tcl

По умолчанию изменение размеров разрешено, т. е. параметр `propagate` установлен в `true`.

Каждые три итерации (т. к. у нас допускается только три виджета в ряде) запускается команда `grid` для трех фреймов с номерами `$i – 2`, `$i – 1` и `$i`: устанавливаются их координаты с помощью параметров `-padx` и `-pady`. Таким образом мы добиваемся упорядочивания элементов таблицы. Результат проделанной работы представлен на рис. 28.1.

28.1.2. Абсолютное размещение

При абсолютном позиционировании мы должны явно указать координаты виджета в сетке. Для этого мы используем параметры `-row` и `-column` при размещении виджета. В листинге 28.2 представлен второй вариант нашей программы: результат будет тем же, но программа использует абсолютное позиционирование для размещения виджетов. Как видите, в программе появился счетчик ряда `row`, который используется как значение параметра `-row` команды `grid`. Может, программа и стала нагляднее, но зато ее код стал больше.

Листинг 28.2. Абсолютное позиционирование с помощью `grid`

```
wm title . "Сетка 2"

set row 1
for {set i 1} {$i <= 12} {incr i} {
frame .fr$i -bg white -width .50i -height .50i
label .l$i -bg blue -fg white -text $i
grid propagate .fr$i false
grid .l$i -in .fr$i
if {![expr $i % 3]} {
       grid .fr[expr $i – 2] -row $row -column 0 -padx 5 -pady 5
       grid .fr$i -row $row -column 2 -padx 5 -pady 5
       grid .fr[expr $i – 1] -row $row -column 1 -padx 5 -pady 5
       incr row
}
}
```

Дабы вы не сомневались в работоспособности кода, результат работы этой программы изображен на рис. 28.2.

Для закрепления навыков абсолютного позиционирования давайте создадим сетку 4×3 и изменим порядок нумерации надписей и фреймов так, чтобы сетка выглядела, как представлено на рис. 28.3. Код программы представлен в листинге 28.3.

Листинг 28.3. Программа grid3.tcl

```
wm title . "Сетка 3"

set counter 1
```

```
for {set i 1} {$i <= 12} {incr i} {
frame .fr$i -bg white -width .50i -height .50i
label .l$i -bg blue -fg white -text $i
grid propagate .fr$i false
grid .l$i -in .fr$i
if {![expr $i % 3]} {
       grid .fr[expr $i — 2] -column $counter -row 0 -padx 5 -pady 5
       grid .fr[expr $i — 1] -column $counter -row 1 -padx 5 -pady 5
       grid .fr$i -column $counter -row 2 -padx 5 -pady 5
incr counter
}
}
```

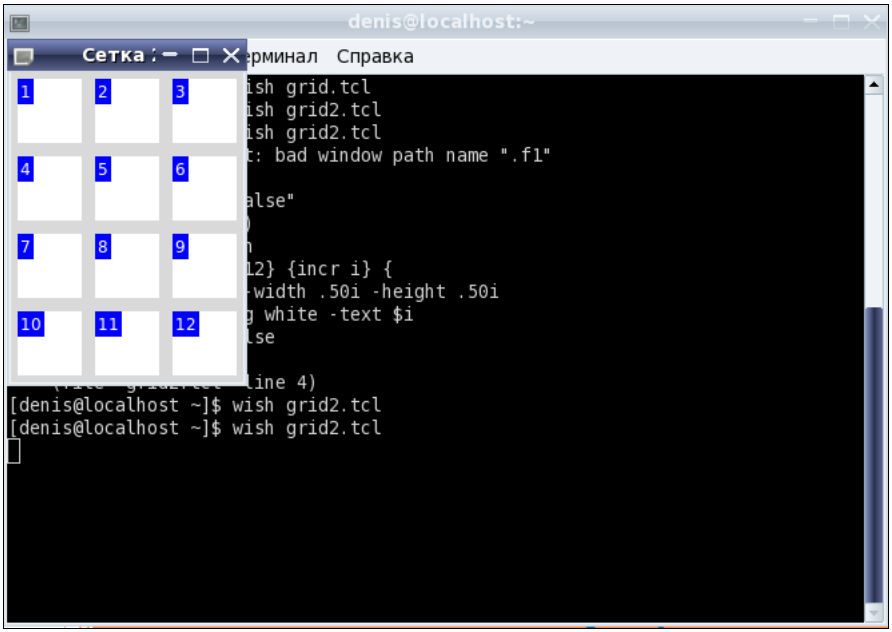


Рис. 28.2. Программа grid2.tcl

Рис. 28.3. Программа grid3.tcl

28.1.3. Объединение ячеек

Если вы когда-нибудь создавали Web-страницу, то наверняка знаете, как с помощью языка HTML объединить ячейки. Объединение ячеек необходимо или для размещения в них больших виджетов (чтобы окно не выглядело уродливым из-за растянутых ячеек) или для улучшения визуализации самой таблицы (если вы выводите информацию в виде таблицы).

Аналогично HTML, в Tk вы можете использовать параметры `-rowspan` и `-columnspan` команды `grid`:

- `-rowspan` — устанавливает число ячеек, которые должны быть объединены по вертикали;
- `-columnspan` — устанавливает число ячеек, которые нужно объединить по горизонтали.

Рассмотрим небольшой пример, демонстрирующий работу этих параметров. С помощью команды `grid` мы создадим небольшую таблицу, где объединим ячейки по вертикали и горизонтали. Чтобы сразу понять, как работают `-rowspan` и `-columnspan`, посмотрите на листинг 28.4, а затем на рис. 28.4, где представлен результат работы программы.

Листинг 28.4. Программа span.tcl

```
label .l1 -bg red -width 10 -height 2 -relief groove -text "Label 1"
label .l2 -bg red -width 10 -height 2 -relief groove -text "Label 2"
label .l3 -bg red -width 10 -height 2 -relief groove -text "Label 3"
label .l4 -bg red -width 10 -height 2 -relief groove -text "Label 4"
label .l5 -bg red -width 10 -height 2 -relief groove -text "Label 5"
label .l6 -bg red -width 10 -height 2 -relief groove -text "Label 6"

grid .l1 -row 0 -column 0 -columnspan 2 -sticky nsew
grid .l2 -row 0 -column 2 -rowspan 2 -sticky nsew
grid .l3 -row 1 -column 0 -columnspan 2 -sticky nsew
grid .l4 .l5 .l6 -sticky nsew
```

Рис. 28.4. Программа span.tcl

В программе вам все должно быть понятным, кроме параметра `-sticky`, о котором раньше не упоминалось. Если виджет меньше размером ячейки сетки (как в нашем случае — размеры надписи меньше размеров сетки), то параметр `-sticky` позволяет разместить виджет внутри ячейки. Параметр `-sticky` определяет стиль размещения

виджета. Стиль задается буквами `n`, `s`, `e` и `w` (можно комбинировать наборы букв, а можно указать только одну букву). Каждая из букв означает одну из сторон ячейки (`n` — север, `s` — юг, `e` — восток, `w` — запад), к которой будет "приклеен" виджет. Если строка стиля содержит символы двух противоположных сторон (`n` и `s` или `e` и `w`), виджет будет увеличен так, чтобы заполнить всю высоту или ширину ячейки. Получается, что параметр `-sticky` заменяет сразу две опции менеджера геометрии pack: `-anchor` и `-fill`. По умолчанию стиль размещения не задан, поэтому виджет будет размещен в центре ячейки и будет иметь свой обычный размер.

28.2. Фреймы

Виджет `frame` (фрейм) выполняет роль контейнера для других виджетов. Вы можете поместить виджеты в фрейм, а затем разместить фрейм в сетке, используя команду `grid`. Фрейм также можно использовать в качестве разделителя между виджетами. Еще одна причина использования фреймов — трехмерные эффекты, позволяющие сделать интерфейс окна более привлекательным. Эффект фрейма задается опцией `-relief`. Программа frames.tcl (листинг 28.5) демонстрирует трехмерные эффекты фреймов.

Листинг 28.5. Программа frame.tcl: демонстрация трехмерных эффектов фреймов

```
frame .f1 -width 1i -height .25i -relief raised -borderwidth 2
label .l1 -text "raised"

frame .f2 -width 1i -height .25i -relief flat -borderwidth 2
label .l2 -text "flat"

frame .f3 -width 1i -height .25i -relief sunken -borderwidth 2
label .l3 -text "sunken"

frame .f4 -width 1i -height .25i -relief groove -borderwidth 2
label .l4 -text "groove"

frame .f5 -width 1i -height .25i -relief solid -borderwidth 2
label .l5 -text "solid"

frame .f6 -width 1i -height .25i -relief ridge -borderwidth 2
label .l6 -text "ridge"

set row 0
foreach num {1 2 3 4 5 6} {
grid .f$num
grid .l$num -in .f$num -sticky nsew
grid propagate .f$num false
grid rowconfigure . $row -pad 5
```

```
incr row
}
grid columnconfigure . 0 -pad 5
```

Сначала программа создает шесть фреймов (потому что есть шесть эффектов) с разными трехмерными эффектами (с разными значениями параметра `-relief`, во всем остальном фреймы одинаковые) и именами `.fN`, где `N` — номер фрейма. Затем программа создает шесть надписей с текстом, поясняющим эффект.

В цикле `foreach` все фреймы помещаются в сетку и выполняется привязка надписей к соответствующим фреймам. Результат приведен на рис. 28.5. Как видите, фреймы позволяют сделать интерфейс окна более привлекательным.

Рис. 28.5. Программа frame.tcl

28.3. Создание окон

Наконец-то мы добрались до самого интересного — до создания окон. Сейчас мы напишем программу, создающую дочернее окно. Программа (листинг 28.6) будет очень проста и не будет выполнять каких-либо полезных действий. Сразу после запуска вы увидите окно с двумя кнопками: **Окно** и **Выход**. Первая кнопка, как вы уже догадались, будет открывать дочернее окно, содержащее только надпись и кнопку закрытия (рис. 28.6).

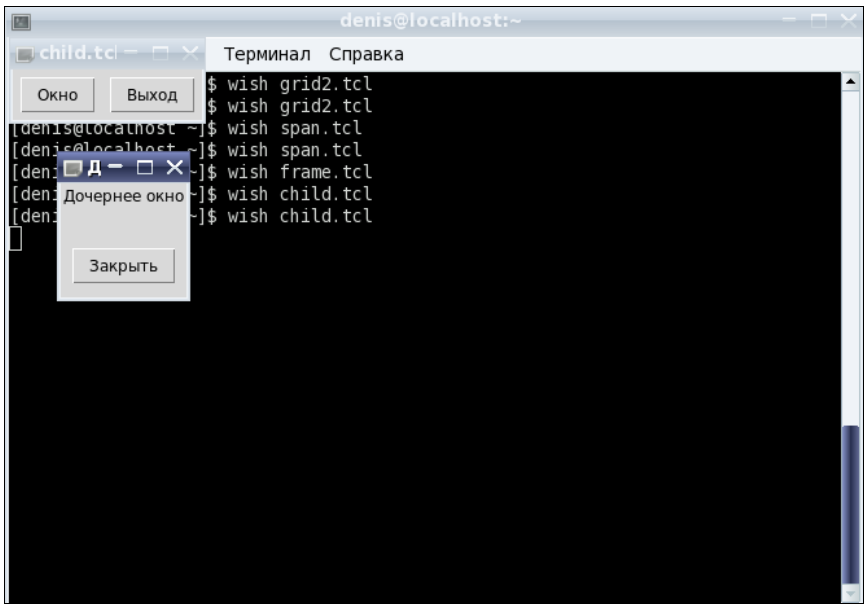


Рис. 28.6. Программа child.tcl

Дополнительные окна можно создать командой `toplevel`:

```
toplevel имя [параметры]
```

Затем порядок работы с окном такой же, как и в случае с основным окном программы: вы добавляете на созданное окно виджеты, используя один из менеджеров геометрии, устанавливаете реакции на события окна и т. д.

Листинг 28.6. Открытие дополнительного окна

```
proc ShowChild {} {
toplevel .w
wm title .w "Дочернее окно"
label .w.l -justify left -height 1 -width 0 -text "Дочернее окно"
button .w.b -text "Закрыть" -command {wm withdraw .w}
grid .w.l -sticky w
grid .w.b -sticky s -pady {30 10}
}

set win [button .win -text "Окно" -command {ShowChild}]
set ext [button .ext -text "Выход" -command exit]
grid $win $ext -padx 5 -pady 5
```

Реакция на нажатие кнопки **Окно** — вызов процедуры `ShowChild`. Процедура `ShowChild` выполняет следующие действия:

- создает окно `.w`;
- изменяет заголовок окна `.w`;
- добавляет надпись с именем `.w.l`;
- добавляет кнопку закрытия дочернего окна (обратите внимание на команду, выполняющуюся при нажатии кнопки);
- размещает надпись и кнопку в сетке окна с помощью `grid`.

28.4. Сообщения

Довольно часто возникает необходимость сообщить что-то пользователю или же запросить подтверждение выполнения операции у пользователя. Создавать информационное окно и диалог с помощью `toplevel` не хочется — незачем изобретать велосипед. И правильно: все уже сделано за вас. Если вам нужно сообщить пользователю об успешном копировании файла (или не очень успешном), используйте команду `message`. А если нужно отобразить стандартный диалог с кнопками **Да** и **Нет**, используйте команду `tk_mesagebox`.

Первая команда применяется для отображения простеньких окон с текстовым сообщением даже без кнопки закрытия окна — ее нужно "доделывать" самостоятельно. Впрочем, мне лень этим заниматься, поскольку есть команда `tk_messageboх`. Рассмотрим, как нужно использовать `message`:

```
message .m -text "Простое сообщение\nПример работы message"
grid .m
```

Честно говоря, результат работы `message` (рис. 28.7) меня не порадовал.

Команда `tk_messageBox` — совсем другое дело. Опции этой команды приведены в табл. 28.1.

Таблица 28.1. *Опция команды* `tk_messageBox`

Опция	Описание
`-default имя`	Задает имя кнопки по умолчанию (`ok`, `cancel`). Имена кнопок приведены в описании опции `-type`. Если параметр `-default` не задан, то кнопки по умолчанию не будет
`-icon пиктограмма`	Выводит пиктограмму возле текстового сообщения. Вы можете использовать пиктограммы `error`, `info`, `question` или `warning` в зависимости от типа сообщения. Если параметр не задан, пиктограмма не отображается
`-message текст`	Текст сообщения
`-parent окно`	Задает родительское окно. Окно сообщения будет отображено поверх родительского окна
`-title заголовок`	Задает текст заголовка окна. По умолчанию используется пустая строка
`-type тип`	Тип окна (определяет набор кнопок): • `abortretryignore` — окно с кнопками **Abort**, **Retry** и **Ignore**; • `ok` — окно с кнопкой **OK**; • `okcancel` — окно с кнопками **OK** и **Cancel**; • `retrycancel` — окно с кнопками **Retry** и **Cancel**; • `yesno` — окно с кнопками **Yes** и **No**; • `yesnocancel` — окно с кнопками **Yes**, **No**, **Cancel**

Рассмотрим небольшой пример использования этой команды:

```
set answer [tk_messageBox -message "Удалить?" -type yesno -icon question]
case $answer {
yes {tk_messageBox -message "Файл удален!" -type ok}
no { exit }
}
```

Результат работы этой команды изображен на рис. 28.8.

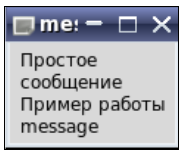


Рис. 28.7. Результат работы `message`

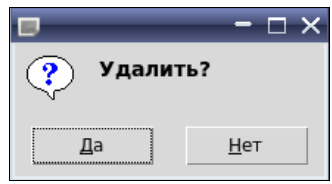


Рис. 28.8. Результат работы команды `tk_messageBox`

28.5. Диалоги открытия и сохранения файла

Для создания диалогов открытия и сохранения файлов используются команды `tk_getOpenFile` и `tk_getSaveFile`:

```
tk_getOpenFile опции
tk_getSaveFile опции
```

Опции диалогов приведены в табл. 28.2.

Таблица 28.2. *Опции диалогов открытия и сохранения файлов*

Опция	Описание
`-defaultextension расширение`	Задает расширение имени файла по умолчанию. Расширение по умолчанию будет автоматически добавлено к имени файла, если оно не имеет расширения. По умолчанию значение этого параметра — пустая строка
`-filetypes список_типов_файлов`	Позволяет сформировать список расширений файлов. Например, если вы создаете графический редактор (или просто программу манипуляции с изображениями), в этот список целесообразно добавить поддерживаемые программой типы файлов. Позже будет показано, как это сделать
`-initialdir каталог`	Задает начальный каталог для диалога открытия/сохранения файла
`-initialfile имя_файла`	Начальное имя файла, которое должно выводиться в диалоге сохранения файла. Команда `tk_getOpenFile` игнорирует этот параметр
`-parent окно`	Диалог появится поверх указанного окна
`-title заголовок`	Задает заголовок диалога

Теперь рассмотрим пример использования диалога:

```
set types {
{{Изображения JPEG} {.jpg} }
{{Изображения GIF} {.gif} }
{{Изображения PNG} {.png} }
{{Все файлы} * }
}

set filename [tk_getOpenFile -filetypes $types -initialdir ~]

if {$filename != ""} {
# Если filename — не пустая строка, открываем файл
}
```

Диалог открытия/сохранения файла не выполняет, собственно, сами операции открытия и сохранения. Он позволяет пользователю выбрать имя файла, которое и

будет передано в программу, далее все зависит от программы — она может открыть файл или произвести над ним любую другую операцию, например вообще удалить. Созданный нами диалог отображен на рис. 28.9.

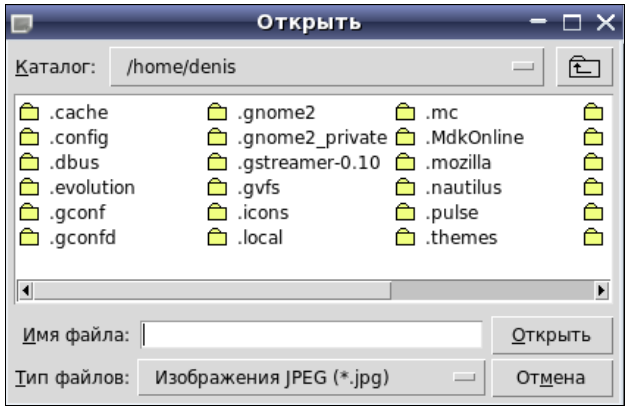


Рис. 28.9. Диалог открытия файла

ГЛАВА 29

Практический пример

29.1. Постановка задачи

В этой небольшой главе мы разработаем простейшую оболочку для популярного сетевого сканера `nmap`. Подобные оболочки создаются для упрощения использования программ без графического интерфейса. Знающие читатели могут заметить, что для `nmap` уже создано несколько графических оболочек. Да, так и есть, зато в этой главе мы создадим свою оболочку, и "по образу и подобию" вы сможете создать аналогичную оболочку для любой текстовой программы.

Графические оболочки создаются не только для начинающих пользователей (сомневаюсь, что начинающий Linux-пользователь будет использовать `nmap`), но и для ленивых квалифицированных пользователей, которым лень вводить одни и те же опции каждый раз при запуске `nmap`.

Синтаксис вызова `nmap` следующий:

```
nmap [опция] адрес_или_диапазон_адресов
```

Опций у `nmap` много, но мы продемонстрируем запуск `nmap` без опций (выполняется стандартное сканирование узла или подсети), с опцией `-O` (попытка определить операционную систему узла) и опцией `-sV` (определение запущенных служб).

Диапазон адресов указывается так:

```
первый_адрес-последний_адрес
```

Например:

```
192.168.0.1-192.168.0.100
```

Нам нужно запустить `nmap` и передать ему опцию и адрес сканируемого компьютера. Наша задача довольно проста. Адрес компьютера пользователь может ввести с помощью текстового поля (команда `entry`), а опцию выбрать с помощью зависимых переключателей (команда `radiobutton`). У нас будет всего три таких переключателя. При желании количество опций можно расширить, добавив дополнительные переключатели.

Итак, наша задача ясна: запустить nmap, передав ему параметры: опцию, задающую режим работы и адрес сканируемого компьютера. Однако просто запустить nmap — мало. Так как мы создаем полноценную оболочку, то нам нужно не только запустить nmap, но и получить его вывод (результат сканирования) в переменную. Что потом с ней делать? Можно просто вывести на консоль с помощью puts, но это не интересно. Гораздо интереснее использовать виджет text, с которым, я надеюсь, вы уже успели познакомиться (вам ведь было поручено разобраться с text самостоятельно — в качестве домашнего задания). Мы получим вывод nmap и загрузим его в виджет text. Перед следующим сканированием мы не будем очищать виджет, что позволит видеть результаты нескольких сканирований.

29.2. Создание оболочки

Запустить nmap и получить его вывод можно с помощью команды exec:

```
exec nmap $options $address
```

Получить вывод в переменную можно так:

```
set nmapOutput [exec nmap $addr $aa]
```

Затем содержимое переменной nmapOutput мы можем отправить на консоль или же в виджет текст:

```
puts $nmapOutput
.t insert end $nmapOutput
```

Чтобы вы лучше представляли, что мы пытаемся создать, посмотрите на рис. 29.1 — на нем изображена уже готовая оболочка.

Начнем создавать нашу "мега-программу". На самом деле программа простая, но мы на ее примере продемонстрируем несколько интересных практических решений.

Первым делом опишем зависимые переключатели:

```
set rNormal [radiobutton .rnormal -text "Обычное сканирование" -value ""]
set rOs [radiobutton .ros -text "Тип ОС" -value "-O"]
set rServices [radiobutton .rservices -text "Запущенные службы" -value "-sV"]
```

Обратите внимание на параметр -value: при выборе переключателя будет передано указанное в нем значение. В нашем случае значением является опция nmap.

Затем мы формируем список радиокнопок с именем radioButtons, создаем переменную aa и надпись-подсказку для текстового поля.

```
set radioButtons [list $rNormal $rOs $rServices]
set aa ""
set labelHint [label .lcolor -text "Адрес или диапазон адресов"]
```

Переменную aa мы свяжем с текстовым полем. Изменение значения aa приведет к изменению содержимого текстового поля и наоборот. Как вы уже догадались, пе-

ременную `aa` мы будем использовать для получения введенного пользователем адреса узла:

```
set Address [entry .address -textvariable aa]
```

Осталось только добавить виджет `text` и кнопки **Сканировать** и **Выход**:

```
set OutputWidget [text .t -setgrid true \
        -width 80 -height 20 -wrap word]

set bSet [button .bset -width 15 -text "Сканировать" \
-command {ScanNmap $selectedButton}]

set bExit [button .bexit -width 15 -text "Выход" -command exit]
```

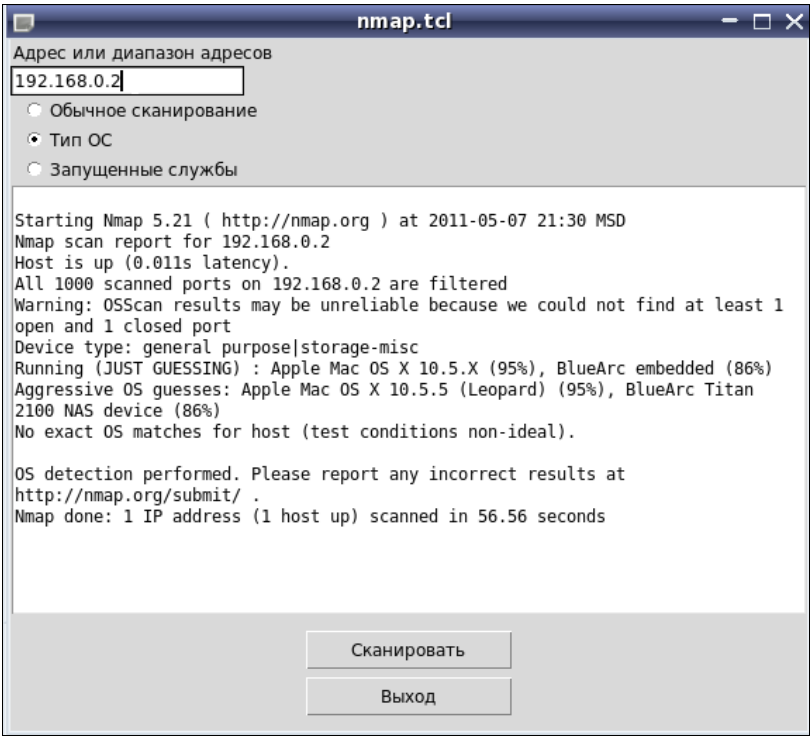

nmap.tcl
Адрес или диапазон адресов
192.168.0.2
Обычное сканирование
Тип ОС
Запущенные службы
Starting Nmap 5.21 (http://nmap.org) at 2011-05-07 21:30 MSD
Nmap scan report for 192.168.0.2
Host is up (0.011s latency).
All 1000 scanned ports on 192.168.0.2 are filtered
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose|storage-misc
Running (JUST GUESSING) : Apple Mac OS X 10.5.X (95%), BlueArc embedded (86%)
Aggressive OS guesses: Apple Mac OS X 10.5.5 (Leopard) (95%), BlueArc Titan 2100 NAS device (86%)
No exact OS matches for host (test conditions non-ideal).
OS detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 56.56 seconds
Сканировать
Выход

Рис. 29.1. Оболочка для `nmap`

При нажатии кнопки **Сканировать** вызывается процедура `ScanNmap`, которая выглядит так:

```
proc ScanNmap {opt} {
global aa;
set nmapOutput [exec nmap $opt $aa]
#puts $nmapOutput
.t insert end $nmapOutput
}
```

Параметр `opt` — это значение выбранного переключателя, опция для `nmap`. Адрес сканируемого узла мы получаем из глобальной переменной `aa`. Далее мы получаем вывод `nmap` и добавляем его в виджет `.t`. Как видите, все довольно просто. Осталось выбрать по умолчанию первый переключатель и "упаковать" все виджеты:

```
$rNormal select
pack $labelHint -anchor w
pack $Address -anchor w
...
```

Полный исходный код программы представлен в листинге 29.1.

Листинг 29.1. Исходный код программы nmap.tcl

```
proc ScanNmap {opt} {
global aa;
set nmapOutput [exec nmap $opt $aa]
#puts $nmapOutput
.t insert end $nmapOutput
}

set rNormal [radiobutton .rnormal -text "Обычное сканирование" -value ""]
set rOs [radiobutton .ros -text "Тип ОС" -value "-O"]
set rServices [radiobutton .rservices -text "Запущенные службы" \
 -value "-sV"]

set radioButtons [list $rNormal $rOs $rServices]
set labelHint [label .lcolor -text "Адрес или диапазон адресов"]

set aa ""
set Address [entry .address -textvariable aa]

set OutputWidget [text .t -setgrid true \
        -width 80 -height 20 -wrap word]

set bSet [button .bset -width 15 -text "Сканировать" \
-command {ScanNmap $selectedButton}]

set bExit [button .bexit -width 15 -text "Выход" -command exit]

$rNormal select

pack $labelHint -anchor w
pack $Address -anchor w
pack $rNormal $rOs $rServices -pady 0 -padx {2 0} -anchor w
pack $OutputWidget
pack $bSet -pady {10 2} -padx 5
pack $bExit -pady {2 10} -padx 5
```

29.3. Запуск оболочки

Поскольку сканирование требует привилегий root, запускать нашу программу нужно с правами root, иначе вместо результата сканирования мы увидим только ругательство `nmap`:

```
sudo wish nmap.tcl
```

Если в вашем дистрибутиве нет команды `sudo`, используйте команду `su`:

```
su -c wish nmap.tcl
```

29.4. Последние штрихи

Нет предела совершенству. Наша программа далека от идеала. Что еще можно добавить? Прежде всего, нужно отдавать себе отчет, что виджет `text` — не резиновый, поэтому добавьте кнопку **Очистить** для удаления результатов предыдущих сканирований.

Также можно добавить дополнительные опции `nmap`, предварительно ознакомившись со справкой `man nmap`.

ЧАСТЬ VII

Библиотека GTK+

Глава 30.	Знакомство с библиотекой
Глава 31.	Первая программа на GTK+
Глава 32.	Виджеты
Глава 33.	Редактор интерфейсов Glade

В предыдущей части книги мы отдохнули от языка C и изучили довольно неплохой язык TCL, а также разобрались, как создать графический интерфейс средствами библиотеки Tk. Настало время рассмотреть популярную библиотеку GTK+, используемую для разработки графического интерфейса на языке C. Также в этой части книги вы познакомитесь с основными функциями библиотеки GLib и редактором интерфейсов Glade.

ГЛАВА 30

Знакомство с библиотекой

30.1. Введение в GTK+

Средствами стандартной библиотеки C графический интерфейс не построишь. А средствами TCL/Tk не создашь серьезную программу. Конечно, привлекательный интерфейс с помощью Tk создать можно — в этом мы уже убедились, но нерасторопность самого TCL не позволит создать большие программные пакеты, поэтому возможности TCL/Tk ограничены написанием небольших утилит. Если хочется создать что-то грандиозное, рекомендуется отдельные, требовательные к ресурсам, участки кода переписать на C для повышения производительности. Но тогда применение TCL/Tk теряет смысл.

Выход, как известно, есть, и заключается он в использовании популярной библиотеки GTK+. Изначально библиотека GTK+ (далее просто GTK) была разработана для использования программой GIMP, отсюда и название — The GIMP Toolkit. Со временем библиотека стала использоваться для разработки других приложений для среды GNOME. Сейчас GTK — основная библиотека графической среды GNOME. Да, вот такое распространение получил самый обычный проект графического редактора, из которого, по сути, выросла целая графическая среда. На основе GTK создана не только GNOME, ее используют также графические окружения LXDE и Xfce.

Библиотека GTK является кросс-платформенной: программу, написанную с использованием GTK, не составит особого труда портировать в Windows. Все зависит от самой программы: главное, чтобы без проблем портировался ее основной код, а все, что относится к GTK, можно с легкостью перенести на другую платформу.

Для запуска GTK-программы на компьютере пользователя нужно, чтобы библиотека GTK была установлена. Как правило, с этим проблем не возникает, поскольку программа GIMP установлена практически на всех Linux-компьютерах (даже если пользователь выбрал графическую среду KDE, использующую библиотеку Qt, он не будет отказываться от GIMP). А набор библиотек для Windows можно скачать по адресу: **http://www.gtk.org/download-windows.html**.

Библиотека GTK базируется на библиотеках GLib и GDK (The GIMP Drawing Kit). Первая оперирует разными типами данных и будет полезна любому приложению, даже не использующему GTK. Вторая взаимодействует с функциями X.Org и используется для построения графических примитивов.

Далее мы познакомимся с библиотекой GLib, которую вы будете применять при создании GTK-приложений. Для использования возможностей этой библиотеки нужно подключить заголовочный файл glib.h.

30.2. Библиотека GLib

30.2.1. Типы данных

При разработке GTK-приложений принято использовать не стандартные типы данных, а типы данных библиотеки GLib. В этой библиотеке вы найдете и типы данных, аналогичные стандартным, и все возможные структуры (деревья, списки), функции обработки ошибок и работы с памятью, а также много чего интересного.

Почему нельзя использовать стандартные типы данных C и стандартные функции распределения памяти? Все это делается, чтобы ваше приложение могло быть с легкостью портировано на другую платформу. Например, на одной платформе обычный тип `int` является 32-разрядным, а на другом — 64-разрядным. Чтобы не ломать себе голову, проще использовать тип `gint` (из библиотеки GLib), а о совместимости позаботится сама библиотека GLib — ведь на другой платформе будет установлена версия GLib, совместимая с этой платформой.

В табл. 30.1 перечислены стандартные типы данных C и их аналоги из библиотеки GLib.

Таблица 30.1. *Соответствие типов данных GLib стандартным типам данных C*

Тип данных C	Тип данных GLib
`char`	`gchar`
`short`	`gshort`
`long`	`glong`
`int`	`gint`
`unsigned char`	`guchar`
`unsigned short`	`gushort`
`unsigned long`	`gulong`
`unsigned int`	`guint`
`float`	`gfloat`
`double`	`gdouble`
`long double`	`gldouble`
`boolean`	`gboolean`
`void*`	`gpointer`

30.2.2. Строки в GLib

В библиотеке GLib есть довольно много функций для работы со строками, но в табл. 30.2 представлены наиболее интересные, на мой взгляд, функции, которые пригодятся любому программисту.

Таблица 30.2. *Некоторые строковые функции библиотеки GLib*

Прототип	Описание
`gchar* g_strchug (gchar* string)`	Удаляет все пробелы в строке `string`, стоящие в начале строки (до первого печатного символа). Функция может пригодиться для контроля введенной пользователем информации
`gchar* g_strchomp(gchar* string)`	Удаляет в строке `string` все пробельные символы (пробелы, символы табуляции и т. д.), стоящие в конце строки
`void g_strdown (gchar* string)`	Переводит все буквы строки `string` в нижний регистр
`void g_strup (gchar* string)`	Переводит все буквы строки `string` в верхний регистр
`void g_strreverse (gchar* string)`	Меняет порядок символов в строке: первый символ становится последним и т. д.
`gchar *g_strconcat (const gchar *string1, ...);`	Создает новую строку, состоящую из строк, переданных ей в качестве параметров. Используется для объединения строк
`gchar *g_strjoin (const gchar *separator, ...);`	Также объединяет строки, но между строками вставляется разделитель, указанный в качестве первого параметра. После первого параметра указываются строки, которые нужно объединить
`gchar *g_strdup (const gchar* string);`	Используется для копирования строк. Создает новую строку, являющуюся копией указанной строки `string`. Возвращает указатель на новую строку
`gchar *g_strndup (const gchar *string, gsize n);`	То же, что и `g_strdup()`, но копируются первые *n* символов строки
`gchar **g_strdupv (gchar **strings);`	Копирует массив строк из `strings`. Чтобы правильно освободить память, занятую этим массивом, нужно использовать функцию `g_strfreev()`
`gchar *g_strcpy (gchar *dest, const gchar* src);`	Копирует содержимое строки из `src` в `dst`. Возвращает указатель на завершающий `\0` в новой строке
`gchar *g_strnfill (gsize length, gchar fill_char);`	Создает новую строку длиной `length` символов, заполненную символом `fill_char`
`void g_strfreev (gchar **strings);`	Освобождает память, занятую массивом строк
`void g_print (const gchar *string);`	Выводит `string` на стандартный вывод

Таблица 30.2 *(окончание)*

Прототип	Описание
`gint g_printf (const gchar *format, ...);`	Аналог стандартной функции `printf()`
`gint g_sprintf (gchar *buf, const gchar *format, ...);`	Аналог стандартной функции `sprintf()`
`gint g_snprintf (gchar *buf, gulong n, const gchar *format, ...);`	Аналог стандартной функции `snprintf()`
`gint g_strcasecmp (const gchar *s1, const gchar *s2);`	Аналог функции `strcasecmp()`
`gint g_strncasecmp (const gchar *s1, const gchar *s2, guint n);`	Аналог функции `strncasecmp()`
`gboolean g_ascii_isalnum (gchar c)`	Возвращает `true`, если `c` — буква алфавита или десятичная цифра
`gboolean g_ascii_isalpha (gchar c)`	Возвращает `true`, если `c` — буква алфавита
`gboolean g_ascii_iscntrl (gchar c)`	Возвращает `true`, если `c` — управляющий символ
`gboolean g_ascii_isdigit (gchar c)`	Возвращает `true`, если `c` — десятичная цифра
`gboolean g_ascii_isgraph (gchar c)`	Возвращает `true`, если `c` — отображаемый символ и не пробел
`gboolean g_ascii_islower (gchar c)`	Возвращает `true`, если `c` — строчная буква
`gboolean g_ascii_isupper (gchar c)`	Возвращает `true`, если `c` — заглавная буква
`gboolean g_ascii_isprint (gchar c)`	Возвращает `true`, если `c` — печатаемый (отображаемый) символ
`gboolean g_ascii_ispunct (gchar c)`	Возвращает `true`, если `c` — символ пунктуации
`gboolean g_ascii_isxdigit (gchar c)`	Возвращает `true`, если `c` — шестнадцатеричная цифра

С остальными строковыми функциями библиотеки GLib вы можете познакомиться по адресу:

http://developer.gnome.org/glib/2.29/glib-String-Utility-Functions.html

30.2.3. Функции распределения памяти

Библиотека GLib также предоставляет собственные аналоги функций распределения памяти. Они полностью аналогичны соответствующим стандартным функциям распределения:

```
gpointer g_malloc( gulong size );
gpointer g_realloc( gpointer mem, gulong size );
void g_free( gpointer mem );
```

30.2.4. Списки

Библиотека GLib содержит средства для работы с одно- и двусвязными списками. В заголовочном файле gslist.h (GLib Single List) описаны средства для работы с односвязными списками, а в файле glist.h — для работы с двусвязным списком.

Особенность двусвязного списка заключается в том, что мы можем передвигаться в двух направлениях по списку — назад и вперед.

Рассмотрим структуры одно- и двусвязного списков:

```
// Односвязный список
typedef struct _GSList GSList;
struct _GSList
{
gpointer data;
GSList *next;                // Указатель на следующий элемент списка
}

// Двусвязный список
typedef struct _GList GList;
struct _GList
{
gpointer data;
GList *next;                 // Указатель на следующий элемент списка
GList *prev;                 // Указатель на предыдущий элемент списка
}
```

Поле `data` предназначено для хранения данных списка, причем они могут быть любого типа, поскольку `gpointer` — это тип `void*`.

Давайте разберемся, как работать со списками. Первым делом нужно объявить список:

```
GList *list = NULL;
GSList *slist = NULL;
```

После нужно добавить элементы в список. Для этого используется две функции — `g_list_append()` или `g_slist_prepend()` — в зависимости от используемого типа списка:

```
gchar *element1 = g_strdup("первый элемент");
list = g_list_append(list, element1);
```

Функции `g_list_append()` и `g_slist_prepend()` добавляют элемент в конец списка. Если нужно добавить элемент в начало списка, используются функции:

```
g_list_prepend(GList *list, gpointer data)
g_slist_prepend(GSList *list, gpointer data)
```

Для вставки нового элемента в определенную позицию применяют функции:

```
GList *g_list_insert( GList *list, gpointer data, gint position )
GSList *g_slist_insert( GSList *list, gpointer data, gint position )
```

Здесь `position` — это номер элемента, перед которым нужно вставить новый элемент. Если `position` равен 0, то элемент будет добавлен в начало списка.

Для удаления элемента используются функции:

```
GList *g_list_remove(GList *list, gpointer data )
GSList *g_slist_remove(GSList *list, gpointer data)
```

Для передвижения по списку используются функции:

- `g_list_nex()`, `g_slist_next()` — на "шаг" вперед;
- `g_list_prev()` — назад.

Далее представлен небольшой пример работы со списком — вывод на консоль всех его элементов.

```
// Объявляем список
GList *list = double_list;

// Добавляем в него элементы соответствующими функциями
...

// Выводим элементы на консоль
while (list!=NULL)
{
g_printf("%s\n",list->data);
list = g_list_next(list);
}
```

По окончании работы со списком не забудьте освободить память:

```
void g_list_free(GList *list);
void g_slist_free(GSList *slist);
```

Отсортировать список можно следующей функцией:

```
GSList *g_slist_sort(GSList * slist, GCompareFunc f);
```

Первый параметр — это список, который нужно отсортировать. Второй — это функция сравнения двух элементов. Вот ее прототип:

```
typedef gint (*GCompareFunc) (gconstpointer a, gconstpointer b);
```

Данную функцию мы должны написать самостоятельно. Она должна принимать два параметра и возвращать целое значение:

- если `a<b`, то –1 (точнее, любое число меньше 0);
- если `a==b`, то 0;
- если `a>b`, то 1 (любое число больше 0).

Библиотека GLib также содержит средства для работы с деревьями — как бинарными, так и произвольными, но мы эти средства рассматривать в этой книге не будем.

30.2.5. Использование таймеров

В своих программах вы можете использовать таймеры для контролирования вычислительных процессов. Средства для работы с таймерами описаны в заголовочном файле gtimer.h.

Создать таймер можно функцией `g_timer_new()`:

```
GTimer *g_timer_new()
```

После создания таймер нужно запустить функцией `g_timer_start()`:

```
GTimer *timer = g_timer_new();
g_timer_start(timer);
```

Функция `g_timer_elapsed()` позволяет узнать время (в секундах), отсчитанное таймером (если нужно узнать количество микросекунд, используется параметр `microseconds`):

```
gdouble g_timer_elapsed( GTimer *timer,
                         gulong *microseconds );
```

Перезапустить таймер можно функцией `g_timer_reset()`:

```
g_timer_reset(GTimer *timer)
```

Для остановки таймера используется функция `g_timer_stop()`, а для его уничтожения — `g_timer_destroy()`:

```
g_timer_stop(GTimer *timer)
g_timer_destroy(GTimer *timer)
```

В листинге 30.1 приведен пример использования таймера. Мы запускаем бесконечный цикл, а время его работы контролируем с помощью таймера. Наш цикл проработает всего 10 секунд. Как и обещал, библиотека GLib оказалась интересной всем программистам, даже если вы не планируете создавать графический интерфейс, таймеры можно использовать для предотвращения зацикливания программы — как дополнительное средство контроля.

Листинг 30.1. Использование таймера

```
#include <stdio.h>
#include <glib.h>
#include <gtimer.h>

int main()
{
gdouble sec;
gulong ms;
gint i;

printf("Пример использования таймера\nОстановка цикла по таймеру");
```

```
// Создаем новый таймер и запускаем его
GTimer *timer = g_timer_new();
g_timer_start(timer);

for (i=1; i>0;)
{
       sec = g_timer_elapsed(timer, &ms);
       g_printf("Итерация %d\n", i);
       if (sec >=10)
        {
              g_timer_stop(timer);
              printf("Таймер остановлен. Мкс: %d\n",ms);
              break;
       }
       i++;
}

g_timer_destroy(timer);

return 0;
}
```

Теперь, когда мы познакомились с основными средствами библиотеки GLib, мы можем приступить к написанию нашей первой GTK-программы, чему посвящена вся следующая глава.

ГЛАВА 31

Первая программа на GTK+

31.1. Виджеты, контейнеры, сигналы и события

Прежде чем приступить к написанию программы, использующей библиотеку GTK+, нужно разобраться с основными понятиями. Кое-что будет вам уже знакомо, если вы прочитали предыдущую часть книги, где рассматривалось создание графического интерфейса средствами Tk.

Виджет — это элемент графического интерфейса: кнопка, список, текстовое поле, меню и т. д. Окно — это тоже виджет. Основным виджетом считается окно. Для размещения виджетов используются контейнеры (окно — тоже контейнер). При программировании на Tk размещением виджетов занимались менеджеры геометрии (`pack`, `grid`), но для более удобного расположения виджетов мы могли поместить их в контейнер — фрейм.

Выравнивать виджеты можно с помощью горизонтальных или вертикальных боксов или с помощью таблиц. Второй способ позволяет более точно расположить виджеты в окне.

Виджет может реагировать на сигналы, например на щелчок мыши. В Tk сигналы назывались событиями, что правильнее, дабы не путать сигналы процессов Linux с событиями графического интерфейса. В GTK кроме понятия сигнал есть еще и понятие *событие*. Например, при нажатии кнопки мыши посылается сигнал `button_press_event`, а с помощью событий можно определить, какая именно кнопка была нажата (например, событие `GDK_1BUTTON_PRESS` говорит о нажатии первой кнопки мыши). Подробно о сигналах и событиях мы поговорим в следующей главе.

Порядок работы с виджетами следующий:

1. Создание виджета специальной функцией из набора библиотеки GTK.
2. Изменение свойств виджета, если это необходимо.
3. Определение сигналов, на которые должен реагировать виджет.
4. Помещение виджета в контейнер.
5. Отображение виджета.

Рассмотрим создание самой обычной кнопки:

```
GtkWidget *button;
...
/* Создаем кнопку с надписью "Привет!" */
button = gtk_button_new_with_label ("Привет!");

/* Устанавливаем реакцию на сигнал:
при нажатии кнопки будет вызвана функция hello() */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                        GTK_SIGNAL_FUNC (hello), NULL);

/* Помещаем кнопку в контейнер — главное окно */
gtk_container_add (GTK_CONTAINER (window), button);

/* Отображаем кнопку */
gtk_widget_show (button);
```

Код вроде бы понятен. До этого кода должны быть объявлены функция `hello()` и виджет `window` (главное окно программы) — нельзя создать кнопку без окна. После создания виджета и установки его свойств не забудьте отобразить виджет с помощью `gtk_widget_show()`, иначе вы его просто не увидите.

31.2. Создание первой программы

Сейчас мы напишем нашу первую GTK-программу: она будет просто выводить пустое окно. Толку от этой программы никакого, зато мы разработаем каркас GTK-программы и разберемся, как ее правильно скомпилировать.

Первым делом нужно подключить заголовочный файл gtk.h:

```
#include <gtk/gtk.h>
```

В функции `main()` нужно объявить виджет окна и инициализировать библиотеку GTK:

```
GtkWidget *window1;
gtk_init( &argc, &argv );
```

После этого можно приступить к созданию виджетов. В листинге 31.1 представлена наша первая GTK-программа.

Листинг 31.1. Первое GTK-приложение

```
#include <gtk/gtk.h>

int main( int   argc, char *argv[] )
{
   GtkWidget *window1;
```

```
    gtk_init( &argc, &argv );
    window1 = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_window_set_title(GTK_WINDOW(window1),"My App");
    gtk_widget_show( window1 );
    gtk_main();

    return 0;
}
```

Функция `gtk_window_new()` создает новое окно, функция `gtk_window_set_title()` — устанавливает заголовок окна. Далее мы отображаем окно и запускаем функцию `gtk_main()`. Функция `gtk_main()` нужна, чтобы наша программа могла реагировать на сигналы. Функции `gtk_init()` и `gtk_main()` должны быть в любой GTK-программе.

31.3. Компиляция программы

Сохраните нашу первую GTK-программу как first.c. Для компиляции программы нужно ввести команду:

```
gcc first.c -o first `gtk-config --cflags` `gtk-config --libs`
```

Команда довольно большая и после каждого внесенного изменения ее лень вводить, поэтому создайте или Makefile, или сценарий оболочки, вызывающий эту команду — как вам больше нравится.

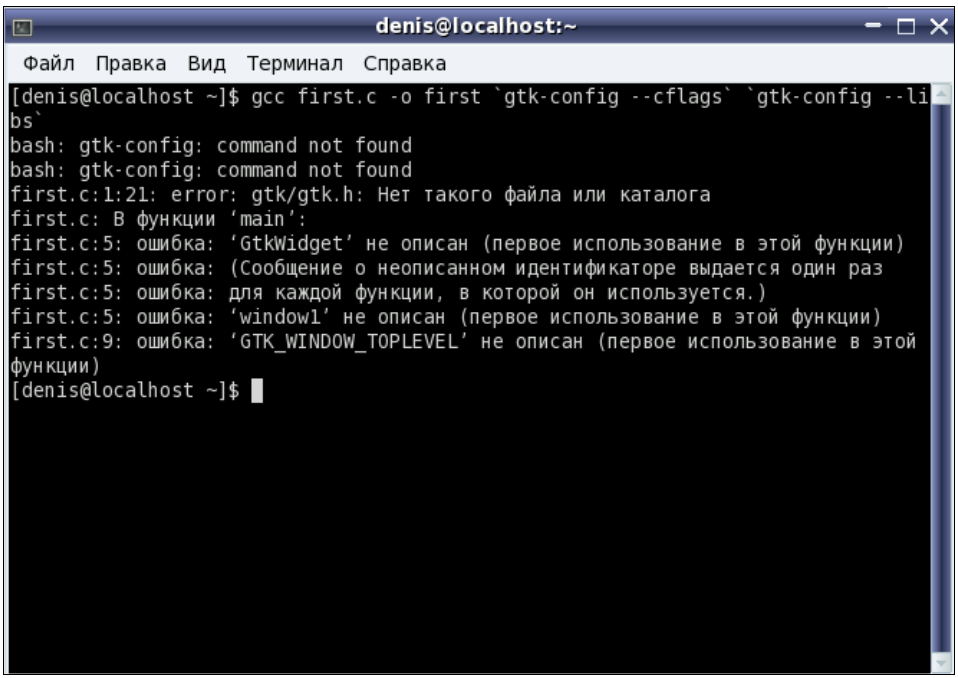

denis@localhost:~
Файл Правка Вид Терминал Справка
[denis@localhost ~]$ gcc first.c -o first `gtk-config --cflags` `gtk-config --li
bs`
bash: gtk-config: command not found
bash: gtk-config: command not found
first.c:1:21: error: gtk/gtk.h: Нет такого файла или каталога
first.c: В функции 'main':
first.c:5: ошибка: 'GtkWidget' не описан (первое использование в этой функции)
first.c:5: ошибка: (Сообщение о неописанном идентификаторе выдается один раз
first.c:5: ошибка: для каждой функции, в которой он используется.)
first.c:5: ошибка: 'window1' не описан (первое использование в этой функции)
first.c:9: ошибка: 'GTK_WINDOW_TOPLEVEL' не описан (первое использование в этой
функции)
[denis@localhost ~]$

Рис. 31.1. Средства разработки для GTK+ не установлены

Скорее всего, если вы еще не установили необходимые пакеты, вы увидите сообщение об отсутствии заголовочного файла gtk.h и программы `gtk-config` (рис. 31.1).

С помощью вашего менеджера пакетов вам нужно установить пакет libgtk+-devel (рис. 31.2), а все дополнительные пакеты будут установлены автоматически как зависимости (рис. 31.3).

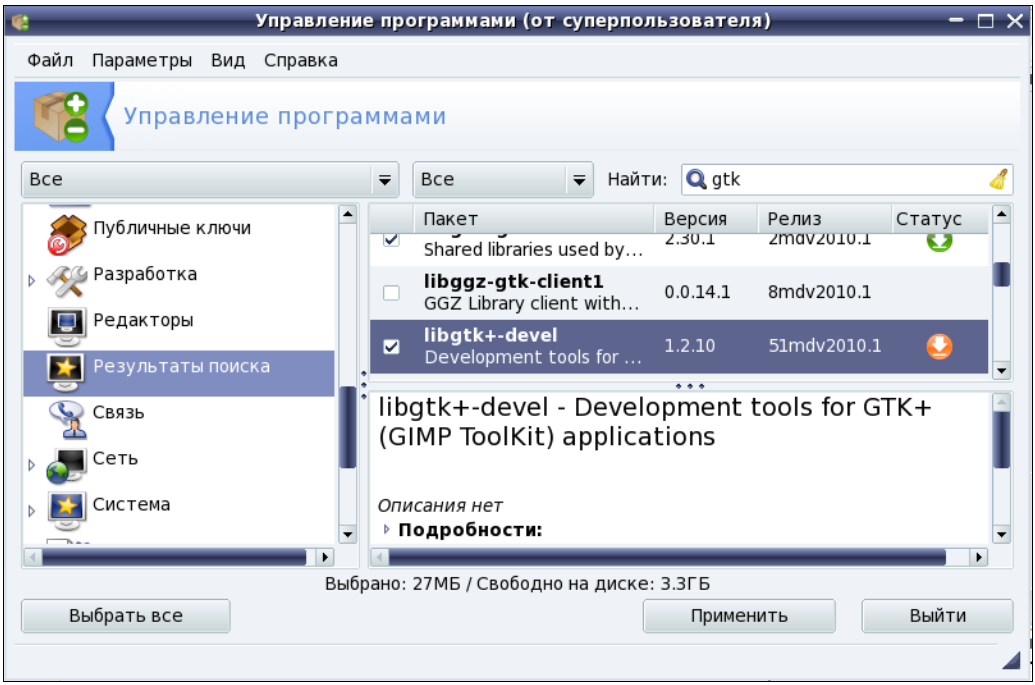


Рис. 31.2. Установка пакета libgtk+-devel

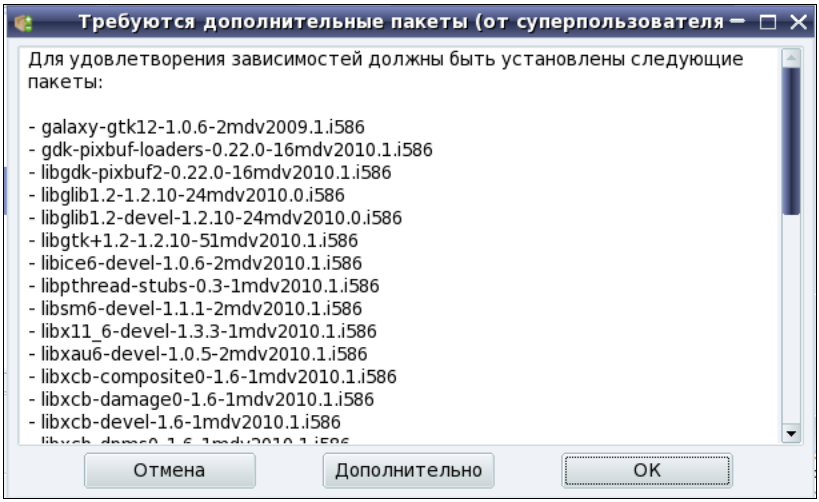


Рис. 31.3. Дополнительные пакеты

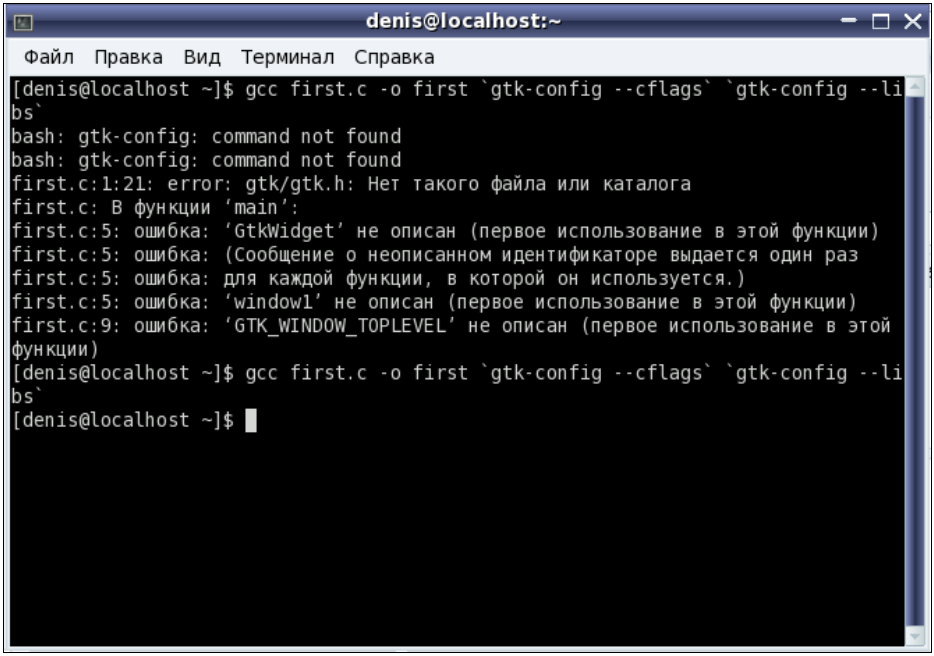

denis@localhost:~
Файл Правка Вид Терминал Справка
[denis@localhost ~]$ gcc first.c -o first `gtk-config --cflags` `gtk-config --li
bs`
bash: gtk-config: command not found
bash: gtk-config: command not found
first.c:1:21: error: gtk/gtk.h: Нет такого файла или каталога
first.c: В функции ‘main’:
first.c:5: ошибка: ‘GtkWidget’ не описан (первое использование в этой функции)
first.c:5: ошибка: (Сообщение о неописанном идентификаторе выдается один раз
first.c:5: ошибка: для каждой функции, в которой он используется.)
first.c:5: ошибка: ‘window1’ не описан (первое использование в этой функции)
first.c:9: ошибка: ‘GTK_WINDOW_TOPLEVEL’ не описан (первое использование в этой
функции)
[denis@localhost ~]$ gcc first.c -o first `gtk-config --cflags` `gtk-config --li
bs`
[denis@localhost ~]$

Рис. 31.4. Программа скомпилирована

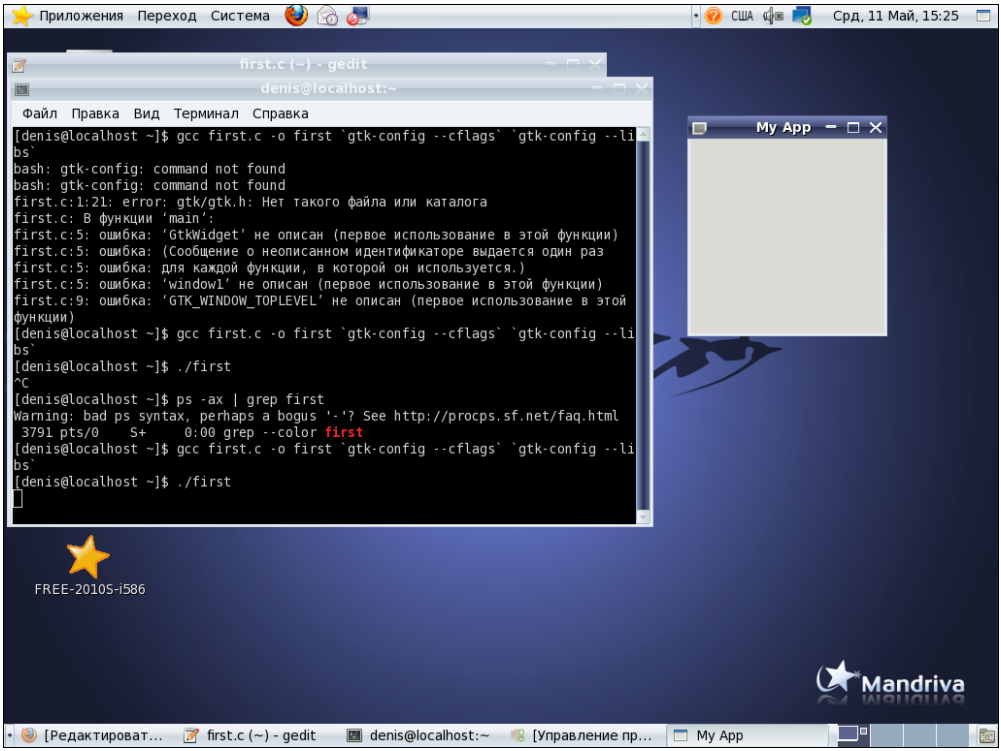


Рис. 31.5. Наша первая GTK-программа

После этого программа скомпилируется без ошибок, о чем свидетельствует рис. 31.4. У вас тоже не должно быть никаких сообщений от компилятора, если вы все сделали правильно.

Запустим программу:

```
./first
```

Вы увидите пустое окно с заголовком **My App** (рис. 31.5).

31.4. Совершенствование программы. Обработчик сигнала

Все гениальное — просто. Наша программа проста, не гениальна (ничего выдающегося в ней нет) и самое интересное, что не идеальна. В этой простой программе уже есть "баг"! Попробуйте закрыть окно программы. Если вы запускали программу из терминала (как я), то заметили, что после закрытия окна программы терминал не был освобожден. Получается, что окно мы закрыли, но процесс завершен, в чем легко убедиться, если ввести команду (рис. 31.6):

```
ps -ax | grep first
```

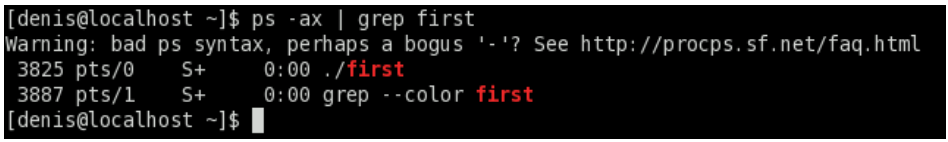


Рис. 31.6. Вывод программы `ps`: процесс `first` не завершен после закрытия окна

Завершить программу придется командой `kill` (`kill <номер процесса>`). Теперь разберемся, почему произошла такая ситуация. Все просто: мы не установили обработчик закрытия окна, поэтому наша программа не знает, что ей нужно делать при закрытии окна. Установим вызов функции `gtk_main_quit()` в качестве обработчика сигнала `destroy` основного окна `window1`:

```
gtk_signal_connect( GTK_OBJECT( window1 ), "destroy",
                    GTK_SIGNAL_FUNC( gtk_main_quit ), NULL );
```

Дабы не тратить попусту бумагу и не переписывать листинг 31.1, добавив в него всего один оператор, поясню, куда нужно вставить вызов `gtk_signal_connect()`: это нужно сделать сразу после создания виджета:

```
window1 = gtk_window_new( GTK_WINDOW_TOPLEVEL );

gtk_signal_connect( GTK_OBJECT( window1 ), "destroy",
                    GTK_SIGNAL_FUNC( gtk_main_quit ), NULL );

gtk_window_set_title(GTK_WINDOW(window1),"My App");

gtk_widget_show( window1 );
```

Перекомпилируйте программу. Вот теперь она завершается при закрытии окна. Обычно вызова `gtk_main_quit()` вполне достаточно, но вы можете указать и свой обработчик закрытия окна. В нем можно выполнить какие-то специальные действия, например закрыть открытые файлы, вывести диалог с подтверждением закрытия окна и т. д. Только не забудьте в конце своего обработчика вызвать функцию `gtk_main_quit()`. Пример определения собственного обработчика закрытия окна приведен в листинге 31.2.

Листинг 31.2. Собственный обработчик закрытия окна

```
#include <gtk/gtk.h>

/* Функция-обработчик закрытия окна */
void close_window1( GtkWidget *widget, gpointer data);

int main( int   argc, char *argv[] )
{
   GtkWidget *window1;

   gtk_init( &argc, &argv );

   window1 = gtk_window_new( GTK_WINDOW_TOPLEVEL );

   gtk_signal_connect( GTK_OBJECT( window1 ), "destroy",
                       (GtkSignalFunc)close_window1, &window1 );

   gtk_widget_show( window1 );

   gtk_main();

   return( 0 );
}

/* Наша функция просто вызывает gtk_main_quit() */
void close_window1( GtkWidget *widget, gpointer data)
{
   gtk_main_quit();
}
```

Если окно слишком маленькое для вашей программы, изменить его размеры поможет функция `gtk_window_set_default_size()`:

```
void gtk_window_set_default_size(GtkWindow *window,
     gint width,
     gint heigth);
```

Функция устанавливает ширину окна `window` равной `width`, а высоту — `heigth`.

Изменить позицию окна можно функцией `gtk_widget_set_uposition()`:

```
void gtk_widget_set_uposition(GtkWindget *widget,
    gint coord_x,
    gint coord_y);
```

Эту же функцию можно использовать для изменения расположения любого виджета — не обязательно окна. Первый параметр — виджет, а остальные два — его координаты.

Функция изменения размера окна объявлена в файле gtk/gtkwindow.h, а функция перемещения — в файле gtk/gtkwidget.h:

```
#include <gtk/gtkwindow.h>
#include < gtk/gtkwidget.h>
...
gtk_window_set_default_size(window1, 400, 400);
gtk_wigdet_set_uposition(window1, 30, 50);
```

В следующей главе вы познакомитесь с остальными виджетами GTK, также в ней будет подробно рассмотрена работа обработчика сигналов графических приложений.

ГЛАВА 32

Виджеты

32.1. Подробно о сигналах

32.1.1. Сигналы и события

Прежде чем приступить к рассмотрению виджетов, обратимся еще раз к функции `gtk_signal_connect()`:

```
gint gtk_signal_connect (GtkObject *object,
                         gchar *name,
                         GtkSignalFunc func,
                         gpointer func_data)
```

Данной функции нужно передать четыре параметра:

- `object` — объект, которому может быть послан сигнал;
- `name` — имя сигнала;
- `func` — имя функции, которая будет вызвана для обработки сигнала;
- `func_data` — данные, которые будут переданы обработчику сигнала.

О любых действиях пользователя ваша программа получает уведомления в виде сигналов. Как только пользователь переместил указатель мыши, программе посылается сигнал о том, что мышь была перемещена. Как только пользователь щелкнул мышью, приложение получает сигнал об этом щелчке и т. д.

Используя функцию `gtk_signal_connect()`, вы можете определить обработчики сигналов, которые будут вызваны в случае получения сигналов вашей программой. Обрабатывать все сигналы может главный виджет — окно, но правильнее будет, если устанавливать реакцию на сигналы для каждого виджета отдельно. Например, вы можете установить реакцию на сигнал `button_press_event`, оповещающий о нажатии кнопки мыши. Но потом вам нужно будет "вычислять", по какому именно виджету щелкнул пользователь. Поэтому проще установить обработку сигналов для каждой кнопки: тогда будет сразу понятно, какую из кнопок нажал пользователь.

Теперь поговорим о самой функции-обработчике сигнала. Данной функции передается два параметра (эти параметры передает библиотека GTK, их не нужно передавать вручную):

- `GtkWidget *widget` — виджет, для которого установлен обработчик сигнала;
- `gpointer data` — данные для обработчика, т. е. содержимое четвертого параметра функции `gtk_signal_connect()`.

В некоторых случаях, в зависимости от типа сигнала, передается три параметра:

- `GtkWidget *widget` — виджет;
- `GdkEvent *event` — событие;
- `gpointer data` — данные.

Рассмотрим самые используемые сигналы:

- `button_press_event` — нажата левая кнопка мыши;
- `button_release_event` — левая кнопка отпущена;
- `motion_notify_event` — движение мыши;
- `delete_event` — удаление объекта;
- `destroy_event` — уничтожение объекта;
- `key_press_event` — нажата клавиша клавиатуры;
- `key_release_event` — клавиша отпущена;
- `enter_notify_event` — указатель мыши вошел в пределы объекта;
- `leave_notify_event` — указатель мыши вышел за пределы объекта;
- `focus_in_event` — объект стал активным (получил фокус);
- `focus_out_event` — объект не активен;
- `drag_begin_event` — начало перемещения объекта;
- `drag_request_event` — запрос на перемещение объекта;
- `drag_end_event` — перемещение объекта;
- `drop_enter_event` — объект перемещен.

Второй параметр (`event`) позволяет более точно определить произошедшее событие:

- `GDK_NOTHING` — не произошло никакого события;
- `GDK_DELETE` — удаление;
- `GDK_DESTROY` — уничтожение;
- `GDK_MOTION_NOTIFY` — уведомление о перемещении;
- `GDK_BUTTON_PRESS` — нажата любая кнопка мыши;
- `GDK_1BUTTON_PRESS` — нажатие первой кнопки мыши;
- `GDK_2BUTTON_PRESS` — нажатие второй кнопки мыши;

- GDK_3BUTTON_PRESS — нажатие третьей кнопки мыши;
- GDK_BUTTON_RELEASE — кнопка (любая) отпущена;
- GDK_KEY_PRESS — нажата клавиша;
- GDK_KEY_RELEASE — клавиша отпущена;
- GDK_ENTER_NOTIFY — указатель мыши в пределах объекта (виджета);
- GDK_LEAVE_NOTIFY — указатель мыши вышел за предела виджета;
- GDK_FOCUS_CHANGE — изменение фокуса ввода;
- GDK_OTHER_EVENT — другое событие.

32.1.2. Виджет *EventBox*

Не все виджеты реагируют на сигналы. Например, надпись (GtkLabel), таблица (GtkTable), горизонтальный и вертикальный контейнеры (GtkHBox, GtkVBox) и некоторые другие не реагируют на сигналы. Чтобы надпись реагировала на щелчок мыши, нужно использовать виджет EventBox. Данный виджет принимает сигналы и позволяет реализовать привязку сигнала к виджету, не реагирующему на сигналы.

Проще всего продемонстрировать работу EventBox на примере. Рассмотрим листинг 32.1, в котором EventBox используется для привязки сигнала button_press_event виджету GtkLabel.

Листинг 32.1. Демонстрация возможностей EventBox

```
#include <gtk/gtk.h>

int main( int argc, char *argv[] )
{
   GtkWidget *window1;              // Главное окно
   GtkWidget *event_box1;           // eventbox
   GtkWidget *label;                // Надпись

    /* Инициализируем GTK */
   gtk_init( &argc, &argv );

   /* Создаем окно */
   window1 = gtk_window_new( GTK_WINDOW_TOPLEVEL );
   gtk_window_set_title( GTK_WINDOW( window1 ), "My App Title" );
  /* Устанавливаем реакцию на закрытие окна */
   gtk_signal_connect(GTK_OBJECT(window1), "destroy",
                                  GTK_SIGNAL_FUNC(gtk_exit), NULL );

  /* Устанавливаем ширину рамки контейнера — окна */
   gtk_container_set_border_width(GTK_CONTAINER(window1), 10);

  /* Создаем event_box */
   event_box1 = gtk_event_box_new();
```

```
 /* Помещаем event_box в контейнер */
   gtk_container_add( GTK_CONTAINER( window1 ), event_box1 );

  /* Отображаем event_box */
   gtk_widget_show( event_box1);

  /* Создаем надпись */
   label = gtk_label_new( "My Label" );

  /* Помещаем надпись в контейнер event_box */
   gtk_container_add(GTK_CONTAINER( event_box1 ), label);

/* Отображаем окно */
   gtk_widget_show(label);

  /* Устанавливаем реакцию GtkLabel на щелчок (при щелчке — выход) */
   gtk_widget_set_events(event_box1, GDK_BUTTON_PRESS_MASK );
   gtk_signal_connect( GTK_OBJECT( event_box1 ), "button_press_event",
                       GTK_SIGNAL_FUNC( gtk_exit ), NULL );

   gtk_widget_realize(event_box1);

  /* Изменяем курсор над надписью — курсор превратится в руку */
   gdk_window_set_cursor( event_box1->window, gdk_cursor_new(GDK_HAND1));

  /* Отображаем окно */
   gtk_widget_show(window1);

   gtk_main();

   return 0;
}
```

Данная программа не только устанавливает реакцию на щелчок для надписи, но и демонстрирует еще несколько интересных возможностей: установку рамки окна, изменение указателя мыши над надписью. Как только указатель мыши будет подведен к надписи, он изменит свой вид на руку — как при подведении указателя к гиперссылке в окне браузера. Возможные типы курсора приведены в табл. 32.1.

Таблица 32.1. *Курсоры GDK*

Название курсора	Как выглядит
GDK_X_CURSOR	
GDK_ARROW	
GDK_BASED_ARROW_DOWN	

Таблица 32.1 *(продолжение)*

Название курсора	Как выглядит
GDK_BASED_ARROW_UP	
GDK_BOAT	
GDK_BOGOSITY	
GDK_BOTTOM_LEFT_CORNER	
GDK_BOTTOM_RIGHT_CORNER	
GDK_BOTTOM_SIDE	
GDK_BOTTOM_TEE	
GDK_BOX_SPIRAL	
GDK_CENTER_PTR	
GDK_CIRCLE	
GDK_CLOCK	
GDK_COFFEE_MUG	
GDK_CROSS	
GDK_CROSS_REVERSE	
GDK_CROSSHAIR	
GDK_DIAMOND_CROSS	
GDK_DOT	
GDK_DOTBOX	
GDK_DOUBLE_ARROW	
GDK_DRAFT_LARGE	
GDK_DRAFT_SMALL	
GDK_DRAPED_BOX	
GDK_EXCHANGE	
GDK_FLEUR	
GDK_GOBBLER	
GDK_GUMBY	
GDK_HAND1	
GDK_HAND2	
GDK_HEART	
GDK_ICON	

Таблица 32.1 *(продолжение)*

Название курсора	Как выглядит
GDK_IRON_CROSS	
GDK_LEFT_PTR	
GDK_LEFT_SIDE	
GDK_LEFT_TEE	
GDK_LEFTBUTTON	
GDK_LL_ANGLE	
GDK_LR_ANGLE	
GDK_MAN	
GDK_MIDDLEBUTTON	
GDK_MOUSE	
GDK_PENCIL	
GDK_PIRATE	
GDK_PLUS	
GDK_QUESTION_ARROW	
GDK_RIGHT_PTR	
GDK_RIGHT_SIDE	
GDK_RIGHT_TEE	
GDK_RIGHTBUTTON	
GDK_RTL_LOGO	
GDK_SAILBOAT	
GDK_SB_DOWN_ARROW	
GDK_SB_H_DOUBLE_ARROW	
GDK_SB_LEFT_ARROW	
GDK_SB_RIGHT_ARROW	
GDK_SB_UP_ARROW	
GDK_SB_V_DOUBLE_ARROW	
GDK_SHUTTLE	
GDK_SIZING	
GDK_SPIDER	
GDK_SPRAYCAN	

Таблица 32.1 *(окончание)*

Название курсора	Как выглядит
GDK_STAR	
GDK_TARGET	
GDK_TCROSS	
GDK_TOP_LEFT_ARROW	
GDK_TOP_LEFT_CORNER	
GDK_TOP_RIGHT_CORNER	
GDK_TOP_SIDE	
GDK_TOP_TEE	
GDK_TREK	
GDK_UL_ANGLE	
GDK_UMBRELLA	
GDK_UR_ANGLE	
GDK_WATCH	
GDK_XTERM	

32.2. Русский текст и GTK

Итак, мы научились устанавливать обработчики событий даже для виджетов, которые не поддерживают данную возможность, и научились изменять вид курсора мыши при подведении его к виджету. Прежде чем рассмотреть другие виджеты, нужно остановиться на еще одном важном моменте.

Если вы заметили, в этой и в предыдущей главе все текстовые надписи (заголовки окон и виджетов) — англоязычные. При попытке создать надпись на русском языке вместо русских букв вы увидите непонятные каракули.

Дабы исправить ситуацию, нужно установить локаль:

```
include <locale.h>
...
setlocale(LC_ALL, "ru_RU.UTF-8");
```

Современные дистрибутивы используют кодировку UTF-8, поэтому приведенный код должен сработать (не забудьте только подключить locale.h). На старых дистрибутивах нужно выбрать KOI8-R:

```
setlocale(LC_ALL, "ru_RU.KOI8-R");
```

На рис. 32.1 приведен наш каркас — простое окно. Как видите, заголовок окна содержит русские буквы. Все это благодаря вызову `setlocale()`. Вызов `setlocale()` нужно сделать до вызова `gtk_init()`:

```
GtkWidget *window1;
setlocale(LC_ALL, "ru_RU.UTF-8");
gtk_init( &argc, &argv );
```

Рис. 32.1. Окно с русскими буквами в заголовке

32.3. Состояния виджета

Управление ресурсами и памятью, необходимыми виджету, выполняется автоматически. Вам ни о чем не нужно заботиться — сразу после создания виджета ему будут предоставлены все необходимые ресурсы. Уничтожение виджета происходит также автоматически — при разрушении (сигнал `destroy`) главного окна программы.

При желании можно уничтожить виджет самостоятельно с помощью функции `gtk_widget_destroy()`, объявленной в gtk/gtkwidget.h:

```
void gtk_widget_destroy (GtkWidget *widget);
```

При уничтожении виджета также уничтожаются все его дочерние виджеты, например при уничтожении окна будут уничтожены все виджеты этого окна.

В наборе функций GTK есть еще одна интересная функция — `gtk_container_remove()`, освобождающая виджет из контейнера:

```
void gtk_container_remove(GtkContainer *cont, GtkWidget *w);
```

Пусть название этой функции не вводит вас в заблуждение: при освобождении виджета происходит его разрушение. Другими словами, у вас не получится извлечь виджет из одного контейнера и добавить в другой. Если виджет вам нужно использовать снова (именно та ситуация, когда нужно переместить виджет между контейнерами), следует создать ссылку на него с помощью функции `g_object_ref()`, а затем извлечь виджет из одного контейнера и поместить в другой. Следующий код перемещает виджет `label` из контейнера `cont1` в контейнер `cont2`:

```
g_object_ref(GTK_WIDGET(label));
gtk_container_remove(GTK_CONTAINER(cont1), label);
gtk_container_add(GTK_CONTAINER(cont2), label);
```

Пока на виджет создана ссылка, вы можете перемещать его между контейнерами. Как только виджет вам будет не нужен, вы можете удалить его с помощью `gtk_widget_destroy()`.

Скрыть виджет можно функцией `gtk_widget_hide()`, а вновь отобразить — функцией `gtk_widget_show()`:

```
void gtk_widget_hide(GtkWidget *w);
void gtk_widget_show(GtkWidget *w);
```

Виджет может находиться в одном из состояний:

- `GTK_STATE_NORMAL` — нормальное;
- `GTK_STATE_ACTIVE` — активное (например, нажата кнопка);
- `GTK_STATE_PRELIGTH` — над виджетом находится указатель мыши;
- `GTK_STATE_SELECTED` — виджет выбран (установлен фокус ввода);
- `GTK_STATE_INSENSITIVE` — виджет не реагирует на ввод (сигналы).

Определить состояние виджета можно так:

```
GTK_WIDGET(w)->state
```

или с помощью макроса `GTK_WIDGET_STATE(widget)`, описанного в файле gtk/gtkwidget.h.

Сделать виджет неактивным можно функцией `gtk_widget_set_sensitive()`:

```
gtk_widget_set_sensitive(widget, FALSE);
```

Данная функция управляет активностью виджета. Если второй параметр равен `FALSE`, виджет станет неактивным, если `TRUE` — активным.

Чтобы наш виджет получил фокус ввода, нужно использовать функцию `gtk_widget_grab_focus()`:

```
gtk_widget_grab_focus(GtkWidget *w);
```

32.4. Контейнеры, поля ввода и кнопки

Наверное, вам надоела теория и хочется немного практики. В этом разделе мы рассмотрим контейнеры, поля ввода и кнопки — этих виджетов будет достаточно для построения простой утилиты. Со всеми этими виджетами гораздо проще разобраться при написании реальной программы — как говорится, лучше один раз увидеть.

Начнем с контейнеров. Контейнеры используются для размещения виджета. Существуют два основных вида контейнера. Первый вид в качестве прародителя использует объект класса `GtkBin`. А второй — объект класса `GtkContainer`. Контейнеры первого вида могут иметь только один дочерний виджет, поэтому они используются для создания специфических интерфейсов: одной кнопки, рамки, окна.

Контейнеры второго вида более функциональны — они могут иметь много дочерних виджетов. Часто используемыми контейнерами являются:

- `GtkHBox` — размещает виджеты горизонтально;
- `GtkVBox` — размещает виджеты вертикально;
- `GtkFixed` — позволяет размещать виджеты в фиксированных координатах;
- `GtkTable` — размещает виджеты в виде таблицы.

Наиболее удачным и простым для восприятия является контейнер `GtkTable`, поэтому на нем и остановимся. Дело в том, что этот контейнер можно использовать при построении программ любой сложности — от обычного диалогового окна до текстового процессора, чего не скажешь о других контейнерах. Даже если вам нужен именно горизонтальный или вертикальный контейнер, вы можете попросту использовать тот же `GtkTable` — никто не мешает вам создать таблицу, состоящую из одной строки или одного столбца.

Вернемся к нашей практической программе. Сейчас мы напишем небольшой конфигуратор, позволяющий редактировать файл /etc/resolv.conf. Напомню вам формат этого файла:

```
search список_доменов
nameserver IP_первого_DNS-сервера
nameserver IP_второго_DNS-сервера
```

Директива `search` определяет наш домен, хотя в ней можно указать список доменов, который будет использоваться для поиска доменного имени. Например, вы можете установить список "com net ru". Когда вы введете в браузере адрес **http://mail**, резолвер (от англ. *resolver* — программа, использующаяся для разрешения доменных имен) сначала попытается разрешить имя узла mail.com, если такое имя не будет доступно, то будет предпринята попытка разрешить имя узла **mail.net** и т. д.

Директивы `nameserver` указывают IP-адреса DNS-серверов, которые будут использоваться при разрешении имен компьютеров. Обычно указывают два IP-адреса (первичный и вторичный серверы), но всего в resolv.conf может быть до четырех директив `nameserver`. В каждой из директив `nameserver` можно указать только один IP-адрес.

Наш конфигуратор не будет вносить изменения в настоящий файл /etc/resolv.conf — для этого нужны права root. Можно, конечно, вызвать `auth` для аутентификации, но мы не будем этого делать, чтобы не усложнять код программы. При желании можно будет потом скопировать файл resolv.conf, сгенерированный нашей программой, в каталог /etc. А можно модифицировать программу так, чтобы она определяла имя запустившего ее пользователя (благо, она это уже умеет) и, если оно не равно root, завершала бы работу. Дальнейшее развитие программы оставлю на ваше усмотрение.

На рис. 32.2 изображена уже готовая программа. Работает она так. Когда пользователь введет что-нибудь в поле ввода и нажмет <Enter>, программа отобразит вве-

денный им текст на консоли. Когда пользователь нажмет **Ok**, введенная им информация будет еще раз выведена на консоль и записана в файл. При нажатии кнопки **Quit** программа завершит свою работу. При нажатии **Ok** программа не завершает работу — это сделано намеренно, дабы можно было экспериментировать с программой. Вы без особых проблем сможете добавить `gtk_main_quit()` в обработчик нажатия кнопки **Ok**.

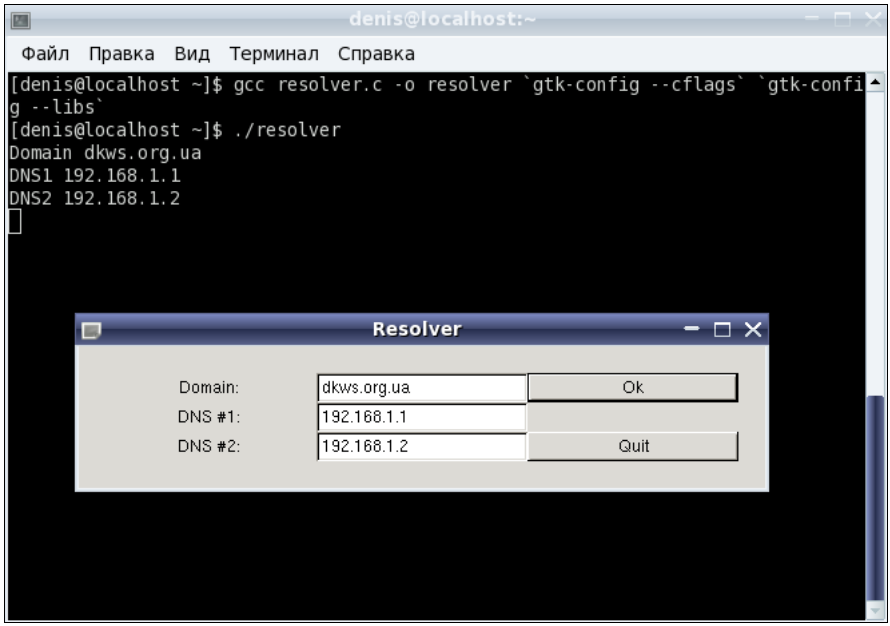


Рис. 32.2. Программа resolver.c

Окно программы достаточно простое и состоит из трех полей ввода, трех надписей и двух кнопок. Поля ввода мы будем хранить в массиве:

```
GtkWidget *edit[3];
```

Создать поле ввода можно функцией `gtk_entry_new()`:

```
edit[i] = gtk_entry_new();
```

После создания поля необходимо вызвать функцию `gtk_entry_set_editable()`, иначе пользователь ничего не сможет ввести в это поле.

```
gtk_entry_set_editable(GTK_ENTRY(edit[i]), 1);
```

Также нужно установить реакцию на нажатие клавиши <Enter> для каждого поля:

```
gtk_signal_connect(GTK_OBJECT(edit[i]), "activate",
                        GTK_SIGNAL_FUNC(enter_callback), edit[i]);
```

По большому счету, в реальной программе реакция на нажатие <Enter> нужна не всегда, но сейчас мы разрабатываем демонстрационную программу, установка обработчика нажатия <Enter> будет очень полезной в общеобразовательных целях.

При нажатии <Enter> будет вызвана функция `enter_callback()`, которая выведет текст поля. Код этой функции будет приведен далее. Этой функции мы передаем не текст поля, а все поле — т. е. виджет целиком. Получить текст поля очень просто:

```
domain = gtk_entry_get_text(GTK_ENTRY(edit[0]));
dns1 = gtk_entry_get_text(GTK_ENTRY(edit[1]));
dns2 = gtk_entry_get_text(GTK_ENTRY(edit[2]));
```

При нажатии кнопки **Ok** программа должна записать полученную от пользователя информацию в файл resolv.conf, который будет находиться в текущем каталоге:

```
/* Перезаписываем файл resolv.conf в текущем каталоге */
if ((resolv = fopen("resolv.conf","w")) == NULL)
{
/* Недостаточно прав, нет места на диске или другая ошибка... */
g_print ("ERR: Cannot to open resolve.conf file\n");
gtk_main_quit ();
}

/* Запись в файл */
fprintf(resolv,"domain %s\n", domain);
fprintf(resolv,"nameserver %s\n", dns1);
fprintf(resolv,"nameserver %s\n", dns2);
fclose(resolv);
```

Теперь поговорим о надписях. У нас всего три надписи, но мы будем экономно использовать системные ресурсы. Обычно создают отдельную переменную для каждого виджета, но в нашем простом случае не вижу в этом смысла. Нам не нужно работать с надписями: перемещать их, обрабатывать сигналы надписей с помощью `EventBox`. Именно поэтому отдельная переменная для каждой надписи не нужна, не нужен и массив надписей. Мы будем работать так: объявим всего одну переменную, затем создадим надпись, поместим ее в контейнер, затем опять создадим надпись с использованием этой же переменной, поместим ее в контейнер и т. д. Код будет выглядеть примерно так:

```
label = gtk_label_new("Domain: ");

gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 1, 0, 1);
gtk_widget_show (label);

label = gtk_label_new("DNS #1: ");
gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 1, 1, 2);
gtk_widget_show (label);

label = gtk_label_new("DNS #2: ");
gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 1, 2, 3);
gtk_widget_show (label);
```

Кнопки (виджет `button`) в GTK могут реагировать на события (сигналы), представленные в табл. 32.2.

Таблица 32.2. *Сигналы кнопок*

Сигналы	Описание
clicked	Щелчок
pressed	Кнопка нажата мышью (и пока не отпущена)
released	Кнопка отпущена
enter	Указатель мыши в пределах кнопки
leave	Указатель мыши вышел за пределы кнопки

Настало время рассмотреть исходный код программы resolver (листинг 32.2). Программа тщательно прокомментирована, а после листинга будет рассмотрен контейнер `GtkTable`.

Листинг 32.2. Программа resolver.c

```
#include <gtk/gtk.h>
#include <stdlib.h>
#include <stdio.h>

/* Строки для хранения доменного имени, IP-адресов DNS-серверов */
gchar *domain, *dns1, *dns2;

/* Массив из трех полей ввода. Первое предназначено для ввода имени
домена, два вторых — [1] и [2] — для ввода IP-адресов серверов DNS*/
GtkWidget *edit[3];

/* Дескриптор файла */
FILE *resolv;

/* Функция записи в файл, вызывается при нажатии Ok */
void writetofile( GtkWidget *widget,
                  gpointer   data )
{
        /* С помощью функции gtk_entry_get_text() мы получаем введенный
        пользователем текст из полей ввода */
        domain = gtk_entry_get_text(GTK_ENTRY(edit[0]));
        dns1 = gtk_entry_get_text(GTK_ENTRY(edit[1]));
        dns2 = gtk_entry_get_text(GTK_ENTRY(edit[2]));

        /* Выводим прочитанный текст на консоль */
        g_print ("Domain %s\n", domain);
        g_print ("DNS1 %s\n", dns1);
        g_print ("DNS2 %s\n", dns2);
```

```
        /* Перезаписываем файл resolv.conf в текущем каталоге */
        if ((resolv = fopen("resolv.conf","w")) == NULL)
        {
                /* Произошла ошибка... */
                g_print ("ERR: Cannot to open resolve.conf file\n");
                gtk_main_quit ();
        }

        /* Запись в файл
        Вместо устаревшей инструкции domain в файл выводится инструкция
        search, хотя на консоль выводится Domain — так понятнее обычному
        пользователю */
        fprintf(resolv,"search %s\n",domain);
        fprintf(resolv,"nameserver %s\n",dns1);
        fprintf(resolv,"nameserver %s\n",dns2);
        fclose(resolv);
}

/* Обработчик закрытия окна или нажатия кнопки Quit */
gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{

        /* Функция gtk_main_quit() используется для завершения
        работы GTK-программы. Не нужно для этого использовать exit()
        из stdlib.h */
            gtk_main_quit ();
            return(FALSE);
}

/* Обработчик нажатия клавиши <Enter> */
void enter_callback( GtkWidget *widget,
                     GtkWidget *entry )
{
        domain = gtk_entry_get_text(GTK_ENTRY(entry));
        printf("Text: %s\n", domain);
}


int main( int   argc,
          char *argv[] )
{

        GtkWidget *window;    /* Окно */
        GtkWidget *button;    /* Кнопка */
        GtkWidget *table;     /* Таблица для размещения виджетов */
        GtkWidget *label;     /* Надпись */
```

```
gint i;                /* Просто счетчик */

/* Инициализация любой GTK-программы */
gtk_init (&argc, &argv);

/* Создаем новое окно */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

/* Устанавливаем заголовок окна */
gtk_window_set_title (GTK_WINDOW (window), "Resolver");

/* Реакция на кнопку закрытия окна. Сигнал — delete_event
Обработчик: функция delete_event(), которая описана ранее
*/
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (delete_event), NULL);

/* Устанавливаем рамку окна */
gtk_container_set_border_width (GTK_CONTAINER (window), 20);

/* Создаем таблицу 3x3 для размещения виджетов*/
table = gtk_table_new (3, 3, TRUE);

/* Помещаем таблицу в контейнер окна. Обязательно! */
gtk_container_add (GTK_CONTAINER (window), table);

/* Рисуем надписи, помещаем их в ТАБЛИЦУ и отображаем.
Нам не нужно объявлять отдельную переменную для
  каждой надписи */

label = gtk_label_new("Domain: ");

/* О координатах ячеек поговорим после этого листинга */
gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 1, 0, 1);
gtk_widget_show (label);

label = gtk_label_new("DNS #1: ");
gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 1, 1, 2);
gtk_widget_show (label);

label = gtk_label_new("DNS #2: ");
gtk_table_attach_defaults (GTK_TABLE(table), label, 0, 1, 2, 3);
gtk_widget_show (label);

/* Заполняем наш массив полей ввода */
for(i=0; i<3; i++)
```

```
        {
                /* Новое поле */
                edit[i] = gtk_entry_new();
                gtk_entry_set_editable(GTK_ENTRY(edit[i]), 1);

                /* Определяем одну для всех реакцию на сигнал activate —
                нажатие клавиши <Enter>*/
                gtk_signal_connect(GTK_OBJECT(edit[i]), "activate",
                                   GTK_SIGNAL_FUNC(enter_callback),
                                   edit[i]);

                /* Помещаем edit[i] в таблицу */
                gtk_table_attach_defaults (GTK_TABLE(table), edit[i],
                                           1, 2, i, i+1);
                /* Показываем поле ввода*/
                gtk_widget_show (edit[i]);
        }

        /* Создаем кнопку Ok, помещаем в таблицу,
        определяем реакцию на нажатие и показываем */
        button = gtk_button_new_with_label ("Ok");
        gtk_table_attach_defaults (GTK_TABLE(table), button, 2, 3, 0, 1);
        gtk_signal_connect(GTK_OBJECT(button),"clicked",
                           GTK_SIGNAL_FUNC(writetofile), NULL);
        gtk_widget_show (button);

        /* Тоже самое для кнопки Quit */
        button = gtk_button_new_with_label ("Quit");
        gtk_table_attach_defaults (GTK_TABLE(table), button, 2, 3, 2, 3);
        gtk_signal_connect(GTK_OBJECT(button),"clicked",
                           GTK_SIGNAL_FUNC(delete_event), NULL);
        gtk_widget_show (button);

        gtk_widget_show (table);     /* Показываем таблицу */
        gtk_widget_show (window);    /* Показываем окно */

        /* Запускаем GTK-программу */
        gtk_main ();

        return 0;
}
```

Думаю, основной код программы понятен. Осталось разобраться с контейнером `GtkTable`. Мы создаем таблицу для размещения виджетов размером 3×3, позволяющую разместить девять виджетов. Компоновка не совсем удачная, но для демонстрации использования таблиц вполне сойдет. Таблица создается следующим оператором:

```
table = gtk_table_new (3, 3, TRUE);
```

Далее нужно разместить в таблице наши виджеты. Все мы играли в морской бой или хотя бы пользовались Excel: принцип такой же. После создания таблицы нужно разместить в ней виджеты:

```
gtk_table_attach_defaults (GTK_TABLE(table), button, 2, 3, 0, 1);
```

Сначала указываются координаты по X, затем — по Y. По горизонтали (X) кнопка **Ok** будет размещена в последнем столбце, между 2 и 3. А по вертикали (Y) — в первом — между 0 и 1. Визуально схему размещения элементов можно изобразить так:

```
0              1         2       3
 Domain  Поле1      Ок
1
 DNS1            Поле2
2
 DNS2            Поле3     Quit
3
```

Думаю, принцип размещения виджетов в таблице ясен. Даже если что-то сейчас окончательно непонятно, скоро все станет на свои места — когда вы начнете использовать средства визуального проектирования интерфейса вроде Glade.

32.5. Зависимые и независимые переключатели

Как мы уже знаем, переключатели бывают двух типов: зависимые (radio buttons) и независимые (check buttons). Переключатели являются кнопками, поэтому для них характерны те же события, что и для кнопок (см. табл. 32.2).

Независимые переключатели создаются функциями:

```
GtkWidget *gtk_check_button_new( void );
GtkWidget *gtk_check_button_new_with_label ( gchar *label );
```

Первая создает переключатель без надписи (если вы хотите указать надпись отдельно), а вторая — с надписью, которая будет отображена, как правило, справа от переключателя. Затем нужно, как обычно, поместить виджеты в контейнер и отобразить. Я считаю, что второй вариант более удобный — редко используется переключатель без сопровождающей надписи, а создавать ее отдельно нет смысла, т. к. есть функция `gtk_check_button_new_with_label()`. Первая функция может понадобиться, если вместо надписи для описания переключателя будет использоваться другой объект, например изображение.

Зависимые переключатели можно создать тоже с помощью двух аналогичных функций:

```
GtkWidget *gtk_radio_button_new( GSList *group );
GtkWidget *gtk_radio_button_new_with_label( GSList *group, ghar *label );
```

Параметр `group` указывает на принадлежность переключателя к группе. В пределах группы активным может быть только один переключатель. Группу можно создать функцией:

```
GSList *gtk_radio_button_group( GtkRadioButton *radio_button );
```

Однако существует другой способ, позволяющий обойтись без переменной группы:

```
button2 = gtk_radio_button_new_with_label(
                gtk_radio_button_group (GTK_RADIO_BUTTON (button1)),
                "button2");
```

С помощью функции `gtk_toggle_button_set_active()` можно сделать одну из кнопок активной:

```
void gtk_toggle_button_set_active( GtkToggleButton *toggle_button, gint state );
```

Никакой ошибки нет: функция `gtk_toggle_button_set_active()` относится к набору для работы с toggle-кнопками. Кнопки такого типа похожи на обычные кнопки, но могут находиться в одном из двух состояний: нажата или нет. Поскольку переключатели — это один из вариантов кнопки, то к ним тоже можно применять функцию `gtk_toggle_button_set_active()`. Во всяком случае, аналогичной функции в наборе для работы с переключателями нет. Получить дополнительную информацию (вместе с примером) о работе с toggle-кнопками можно по адресу:

http://www.gtk.org/tutorial1.2/gtk_tut-6.html

Определить, какой переключатель в группе активен, а какой нет, автоматически невозможно, т. е. нет такой функции, которая бы возвращала идентификатор выбранного переключателя в группе. Существует два варианта проверки состояния переключателя:

1. Обработчик сигнала `clicked` — вы создаете функцию-обработчик сигнала `clicked` для всех переключателей группы. Данная функция будет вызываться, как только кто-то щелкнул на переключателе. В качестве параметра функции будет передаваться нечто, позволяющее идентифицировать выбранный переключатель. Функция устанавливает состояние глобальной переменной в зависимости от выбранного переключателя. Затем в другом участке программы анализируется значение этой переменной и выполняются определенные действия.

2. Проверка состояния функцией `gtk_toggle_button_get_active()` — еще одна функция из набора для toggle-кнопок, которую удобно использовать для проверки, включен или нет переключатель (причем не важно, какой тип у переключателя — check button или radio button). Такой способ более простой в реализации и не требует вызова функции каждый раз при включении/выключении переключателя. Функция `gtk_toggle_button_get_active()` вызывается для проверки состояния конкретного переключателя, т. е. ее нужно вызвать столько раз, сколько переключателей у вас имеется. Пожалуй, это единственный недостаток этого способа, но в целом он более удобный, поэтому его мы и будем использовать.

Проверить состояние переключателя можно так:

```
if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(button1)))
              g_print("Переключатель button1 включен\n");
```

Обратите внимание: чтобы функция `gtk_toggle_button_get_active()` работала правильно и вы не получили сообщение о недопустимом типе параметра, нужно тип переключателя (зависимого или независимого) привести к типу `GTK_TOGGLE_BUTTON`.

Следующий листинг демонстрирует работу с тремя зависимыми переключателями (листинг 32.3). Данный пример также демонстрирует работу с вертикальным контейнером `GtkVBox`. Мы создадим простое окно выбора дистрибутива. В нем будет всего два зависимых переключателя (вы можете создать их сколько угодно) и кнопка **OK**. При нажатии кнопки производится "вычисление" активного переключателя и выводится соответствующее сообщение на консоль, поэтому запускать программу нужно из терминала. После вывода сообщения, соответствующего выбору пользователя, программа завершает работу.

Листинг 32.3. Зависимые переключатели

```
#include <gtk/gtk.h>
#include <glib.h>
#include <locale.h>

/* Виджеты переключателей объявлены как глобальные переменные,
чтобы к ним смогла получить доступ функция close_applicaion */
GtkWidget *button1;
GtkWidget *button2;

/* Проверяет, какой переключатель выбран, и завершает работу
программы */
gint close_application( GtkWidget *widget,
                        GdkEvent  *event,
                        gpointer   data )
{
if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(button1)))
 g_print("Fedora — классика\n");

if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(button2)))
g_print("Denix — хороший выбор\n");

  gtk_main_quit();    /* Завершаем работу программы */

  return(FALSE);
}

int main( int   argc,
          char *argv[] )
```

```
{
/* Виджеты окна, контейнеров, разделителя и кнопки */
GtkWidget *window = NULL;
GtkWidget *box1;
GtkWidget *box2;
GtkWidget *separator;
GtkWidget *button;

/* Идентификатор группы зависимых переключателей */
GSList *group;

/* Поддержка русского языка */
setlocale(LC_ALL, "ru_RU.UTF-8");

gtk_init(&argc,&argv);

window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

/* Устанавливаем обработчик закрытия окна */
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                      GTK_SIGNAL_FUNC(close_application), NULL);

/* Заголовок окна */
gtk_window_set_title (GTK_WINDOW (window), "Выбор дистрибутива");

/* Ширина рамки контейнера */
gtk_container_set_border_width (GTK_CONTAINER (window), 0);

/* Создаем вертикальный контейнер и помещаем его в окно */
box1 = gtk_vbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (window), box1);

/* Отображаем контейнер */
gtk_widget_show (box1);

/* Создаем второй контейнер */
box2 = gtk_vbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

/* Создаем первый переключатель и помещаем его во второй контейнер */
button1 = gtk_radio_button_new_with_label (NULL, "Fedora");
gtk_box_pack_start (GTK_BOX (box2), button1, TRUE, TRUE, 0);
gtk_widget_show (button1);   /* Отображаем виджет */

/* Вычисляем группу */
group = gtk_radio_button_group (GTK_RADIO_BUTTON (button1));
```

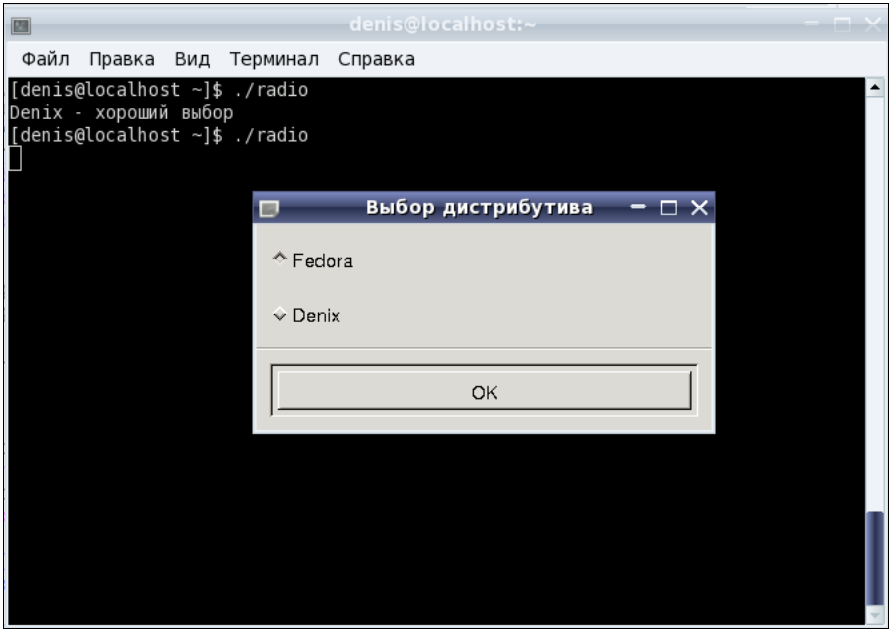


Рис. 32.3. Зависимые переключатели

```
/* Создаем второй переключатель */
button2 = gtk_radio_button_new_with_label(group, "Denix");
gtk_box_pack_start (GTK_BOX (box2), button2, TRUE, TRUE, 0);
gtk_widget_show (button2);

/* Создаем разделитель */
separator = gtk_hseparator_new();
gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
gtk_widget_show (separator);

/* Далее идет упаковка виджетов, хотя можно было бы ее упростить или
вообще использовать GtkTable — к переключателям дальнейший код уже не
относится */
box2 = gtk_vbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);
gtk_widget_show (box2);

/* Создаем кнопку OK с обработчиком close_application */
button = gtk_button_new_with_label ("OK");
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC(close_application),
                           GTK_OBJECT (window));
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
```

```
gtk_widget_grab_default (button);
gtk_widget_show (button);
gtk_widget_show (window);

gtk_main();

return(0);
}
```

Окно программы приведено на рис. 32.3.

32.6. Список *CList*

Виджет `CList` представляет собой список, состоящий из нескольких колонок. Ячейки такого списка могут содержать текстовые значения. Мы можем обратиться отдельно к каждой ячейке списка. Создать список можно одной из функций:

```
GtkWidget *gtk_clist_new ( gint columns );
GtkWidget *gtk_clist_new_with_titles( gint columns, gchar *titles[] );
```

Первая функция создает список без заголовков, а вторая — с заголовками. Параметр `columns` задает количество колонок. Добавить элемент в список позволяют функции:

```
gint gtk_clist_prepend( GtkCList *clist, gchar *text[] );
gint gtk_clist_append( GtkCList *clist, gchar *text[] );
```

Первая функция добавляет новый элемент в начало списка, а вторая — в его конец. Если необходимо вставить элемент в определенную позицию, то нужно использовать функцию:

```
void gtk_clist_insert( GtkCList *clist,
                       gint row, gchar *text[]);
```

Функция позволяет вставить новый элемент в строку `row`. Нумерация строк списка начинается с 0. Для удаления элементов списка можно использовать одну из функций:

```
void gtk_clist_remove( GtkCList *clist, gint row );
void gtk_clist_clear( GtkCList *clist );
```

Первая удаляет строку `row`, а вторая — очищает весь список.

Рассмотрим листинг 32.4, в котором демонстрируется работа со списком `CList`. Листинг тщательно закомментирован, поэтому рекомендую внимательно читать исходный код. Программа создает пустой список с колонками. Кнопка **Добавить** добавляет элементы списка, кнопка **Очистить** — удаляет. Кнопка **Спрятать/отобразить** — скрывает и отображает заголовок списка. Окно программы изображено на рис. 32.4.

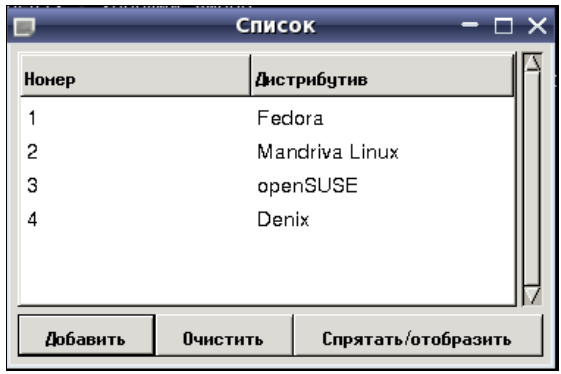


Рис. 32.4. Программа cslist.c

Листинг 32.4. Применение виджета `CList` (cslist.c)

```
#include <gtk/gtk.h>
#include <locale.h>

/* Обработчик кнопки "Добавить", добавляет элементы списка */
void button_add_clicked( gpointer data )
{
    int indx;

    /* Простой список */
    gchar *dist[4][2] = {  { "1", "Fedora" },
                           { "2", "Mandriva Linux" },
                           { "3", "openSUSE" },
                           { "4", "Denix" } };

    for ( indx=0 ; indx < 4 ; indx++ )
        gtk_clist_append( (GtkCList *) data, dist[indx]);

    return;
}

/*Обработчик нажатия кнопки "Очистить" */
void button_clear_clicked( gpointer data )
{
    /* Очищаем список */
    gtk_clist_clear( (GtkCList *) data);
    return;
}

/* Функция прячет/отображает заголовки списка */
void button_hide_show_clicked( gpointer data )
```

```
{
    /*  0 = сейчас видим заголовки  */
    static short int flag = 0;

    if (flag == 0)
    {
        /* Прячем заголовки */
        gtk_clist_column_titles_hide((GtkCList *) data);
        flag++;
    }
    else
    {
        /* Отображаем заголовки */
        gtk_clist_column_titles_show((GtkCList *) data);
        flag--;
    }

    return;
}

/* Данная функция будет вызвана, если пользователь выберет элемент */
void selection_made( GtkWidget      *clist,
                     gint            row,
                     gint            column,
                     GdkEventButton *event,
                     gpointer        data )
{
    gchar *text;

    /* Получаем выбранный текст (элемент списка) */
    gtk_clist_get_text(GTK_CLIST(clist), row, column, &text);

    /* Просто выводим информацию на консоль */
    g_print("Вы выбрали ряд %d. Колонка %d, текст в ячейке %s\n\n",
            row, column, text);

    return;
}

int main( int    argc,
          gchar *argv[] )
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox;
    GtkWidget *scrolled_window, *clist;
    GtkWidget *button_add, *button_clear, *button_hide_show;
```

```
/* Заголовки списка */
gchar *titles[2] = { "Номер", "Дистрибутив" };

setlocale( LC_ALL, "ru_RU.UTF-8");

gtk_init(&argc, &argv);

window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_widget_set_usize(GTK_WIDGET(window), 300, 150);

gtk_window_set_title(GTK_WINDOW(window), "Список");
gtk_signal_connect(GTK_OBJECT(window),
                   "destroy",
                   GTK_SIGNAL_FUNC(gtk_main_quit),
                   NULL);
vbox=gtk_vbox_new(FALSE, 5);
gtk_container_set_border_width(GTK_CONTAINER(vbox), 5);
gtk_container_add(GTK_CONTAINER(window), vbox);
gtk_widget_show(vbox);

/* Создаем окно с полосками прокрутки и упаковываем в него список */
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (
   GTK_SCROLLED_WINDOW(scrolled_window),
   GTK_POLICY_AUTOMATIC,
   GTK_POLICY_ALWAYS);

gtk_box_pack_start(GTK_BOX(vbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);

/* Создаем список с двумя колонками */
clist = gtk_clist_new_with_titles( 2, titles);

/* Обработка выделения */
gtk_signal_connect(GTK_OBJECT(clist), "select_row",
                   GTK_SIGNAL_FUNC(selection_made),
                   NULL);

/* Устанавливаем тень для рамки списка */
gtk_clist_set_shadow_type (GTK_CLIST(clist), GTK_SHADOW_OUT);

/* Устанавливаем ширину для колонки. Колонки нумеруются с 0 */
gtk_clist_set_column_width (GTK_CLIST(clist), 0, 150);

/* Помещаем список в контейнер */
gtk_container_add(GTK_CONTAINER(scrolled_window), clist);
gtk_widget_show(clist);
```

```
    /* Создаем и размещаем кнопки "Добавить", "Очистить",
    "Спрятать/отобразить" */
    hbox = gtk_hbox_new(FALSE, 0);
    gtk_box_pack_start(GTK_BOX(vbox), hbox, FALSE, TRUE, 0);
    gtk_widget_show(hbox);

    button_add = gtk_button_new_with_label("Добавить");
    button_clear = gtk_button_new_with_label("Очистить");
    button_hide_show = gtk_button_new_with_label("Спрятать/отобразить");

    gtk_box_pack_start(GTK_BOX(hbox), button_add, TRUE, TRUE, 0);
    gtk_box_pack_start(GTK_BOX(hbox), button_clear, TRUE, TRUE, 0);
    gtk_box_pack_start(GTK_BOX(hbox), button_hide_show, TRUE, TRUE, 0);

    /* Связываем обработчики */
    gtk_signal_connect_object(GTK_OBJECT(button_add), "clicked",
                              GTK_SIGNAL_FUNC(button_add_clicked),
                              (gpointer) clist);
    gtk_signal_connect_object(GTK_OBJECT(button_clear), "clicked",
                              GTK_SIGNAL_FUNC(button_clear_clicked),
                              (gpointer) clist);
    gtk_signal_connect_object(GTK_OBJECT(button_hide_show), "clicked",
                              GTK_SIGNAL_FUNC(button_hide_show_clicked),
                              (gpointer) clist);

    gtk_widget_show(button_add);
    gtk_widget_show(button_clear);
    gtk_widget_show(button_hide_show);

    gtk_widget_show(window);
    gtk_main();

    return(0);
}
```

32.7. Диалог выбора файлов

Довольно часто приходится писать программы для работы с файлами. В такой программе обязательно будет пункт меню или кнопка **Открыть**, нажав которую пользователь видит диалог открытия файла, позволяющий выбрать файл. Диалог сохранения файла отличается лишь заголовком — **Сохранить файл**.

Сейчас мы напишем небольшую программку, которая вызывает стандартный диалог открытия файла, и запрограммируем обработчики нажатия кнопок **OK** и **Отменить**. При нажатии кнопки **OK** диалог будет выводить на консоль имя выбранного файла, а при нажатии **Отменить** — закроется. Диалог, а также вывод имени файла на консоль изображен на рис. 32.5.

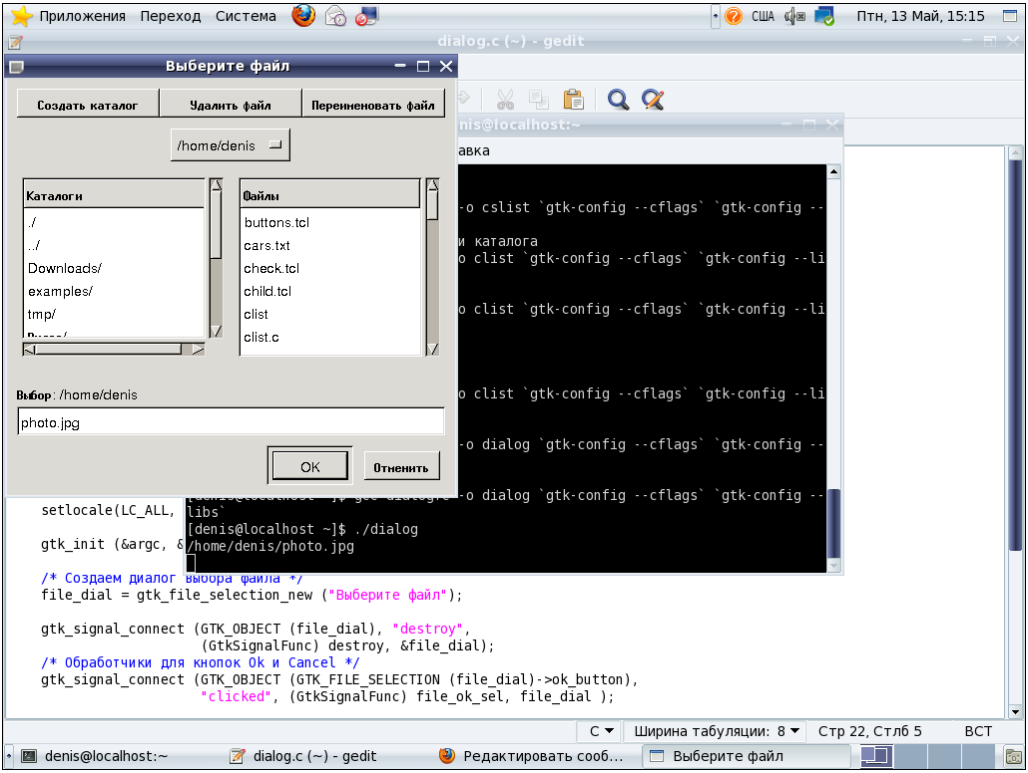


Рис. 32.5. Диалог выбора файла

Листинг 32.9. Диалог выбора файла

```
#include <gtk/gtk.h>
#include <locale.h>

/* Получает имя выбранного файла и выводит его на консоль */
void file_ok_sel( GtkWidget *w, GtkFileSelection *fs )
{
       g_print ("%s\n",
              gtk_file_selection_get_filename (GTK_FILE_SELECTION (fs)));
}

void destroy( GtkWidget *widget, gpointer   data )
{
       gtk_main_quit();
}

int main( int   argc,
          char *argv[] )
{
    GtkWidget *file_dial;
```

```
    setlocale( LC_ALL, "ru_RU.UTF-8");
    gtk_init (&argc, &argv);

    /* Создаем диалог выбора файла */
    file_dial = gtk_file_selection_new ("Выберите файл");

    gtk_signal_connect (GTK_OBJECT (file_dial), "destroy",
                        (GtkSignalFunc) destroy, &file_dial);

    /* Обработчики для кнопок OK и "Отменить" */
    gtk_signal_connect (GTK_OBJECT
       (GTK_FILE_SELECTION (file_dial)->ok_button),
       "clicked", (GtkSignalFunc) file_ok_sel, file_dial );

    gtk_signal_connect_object (GTK_OBJECT (
                    GTK_FILE_SELECTION(file_dial)->cancel_button),
                    "clicked",
                     (GtkSignalFunc) gtk_widget_destroy,
                    GTK_OBJECT (file_dial));

    /* Устанавливает имя файла по умолчанию */
    gtk_file_selection_set_filename (GTK_FILE_SELECTION(file_dial),
                                     "photo.jpg");

    gtk_widget_show(file_dial);
    gtk_main();
    return 0;
}
```

32.8. Визуальная разработка интерфейса пользователя

Мы рассмотрели основы программирования графического интерфейса пользователя на GTK. Остались нерассмотренными многие виджеты — меню, текстовый редактор, выпадающий список и т. д. Однако дальнейшее знакомство с остальными виджетами считаю целесообразным с помощью средств визуального проектирования. Одно из таких средств — редактор интерфейсов Glade. С его помощью вы можете за пару минут "набросать" интерфейс любого окна вашей программы и при этом вам не нужно ломать голову над тем, как виджеты будут расположены в контейнеры, вычислять координаты ячеек GtkTable — как будет выглядеть будущее окно, вы сразу увидите. То же самое касается и свойств виджетов: вам не нужно помнить их наизусть, постоянно читать руководство разработчика. Вы помещаете виджет в контейнер и в правой части окна Glade видите все его свойства, которые можно установить в визуальном режиме. Аналогично можно установить обработчики сигналов, при этом точное название сигнала тоже не обязательно помнить.

Настало время прочитать следующую главу, в которой рассмотрен редактор интерфейсов Glade — программа, позволяющая создать интерфейс пользователя за считанные минуты.

ГЛАВА 33

Редактор интерфейсов Glade

33.1. Быстрая разработка приложений

Наверняка до прочтения этой книги у вас был опыт программирования, пусть не в Linux, но в Windows. Как любой опытный (и даже не очень) программист, вы знакомы хотя бы с одной средой быстрой разработки приложений (RAD, Rapid Application Development). Примерами таких систем являются Microsoft Visual Studio, Borland Delphi, Lazarus и еще ряд других систем.

Преимущество RAD-системы заключается в визуальном проектировании расположения элементов графического окна — виджетов. Вы сразу видите, как будет выглядеть окно вашей программы. Не нужно вычислять расположение виджета, "гадать", почему он расположен не там, где бы вам хотелось, компилировать программу несколько раз, поскольку виджет оказался не в том месте, и т. д.

Для Linux представлено несколько подобных систем: QtDevelop (для разработки приложений, использующих библиотеку Qt), Lazarus (можно сказать, аналог Delphi для Linux) и Glade UI Builder (используется для разработки GTK-приложений).

QtDevelop (в паре с QtDesigner) является практически полноценной RAD-системой, позволяющей не только проектировать интерфейс пользователя, но создавать и отлаживать саму программу (т. е. включает редактор кода и отладчик). То же самое можно сказать и о Lazarus, но эта среда использует язык Pascal (компилятор Free Pascal), в отличие от QtDevelop, где программы пишутся на C++.

Что же касается Glade UI Builder, то он не является RAD-системой в прямом смысле этого слова. Glade — это редактор визуального интерфейса, вы можете с его помощью спроектировать интерфейс пользователя, т. е. расположить виджеты в окне, вы можете определить свойства виджетов, но на этом возможности Glade иссякают. Для написания исходного кода (на C или C++) программы вы должны использовать сторонний редактор, а для отладки — сторонний отладчик, например `gdb`. Glade не обладает функциями ни редактора кода, ни отладчика, даже нет функции вызова соответствующих программ из меню Glade. Тем более Glade не вызывает компилятор для компиляции вашего проекта.

Разработка программы с помощью Glade состоит из следующих этапов:

1. Собственно, сама разработка графического интерфейса. Этот этап выполняется в Glade. По завершении вы получите файл с "расширением" .glade, в котором будет описан интерфейс вашей программы. Формат этого файла стар как мир — самый обычный XML, поэтому при желании данный файл можно редактировать даже в `gedit` — самом обычном текстовом редакторе.
2. После создания файла описания интерфейса нужно создать файл самой программы — на C или C++. В вашей программе вы подключаете Glade и указываете файл интерфейса. Как это сделать, будет показано позже.
3. Когда программа написана, нужно создать Makefile для ее компиляции.
4. Компиляция программы командой `make`.

В обычных RAD-системах все эти этапы выполняются самой средой, в случае с Glade у программиста по-прежнему много ручной работы.

33.2. Установка Glade

Для установки редактора интерфейсов Glade нужно установить пакет, который в зависимости от дистрибутива может называться glade3, glade-gnome или просто

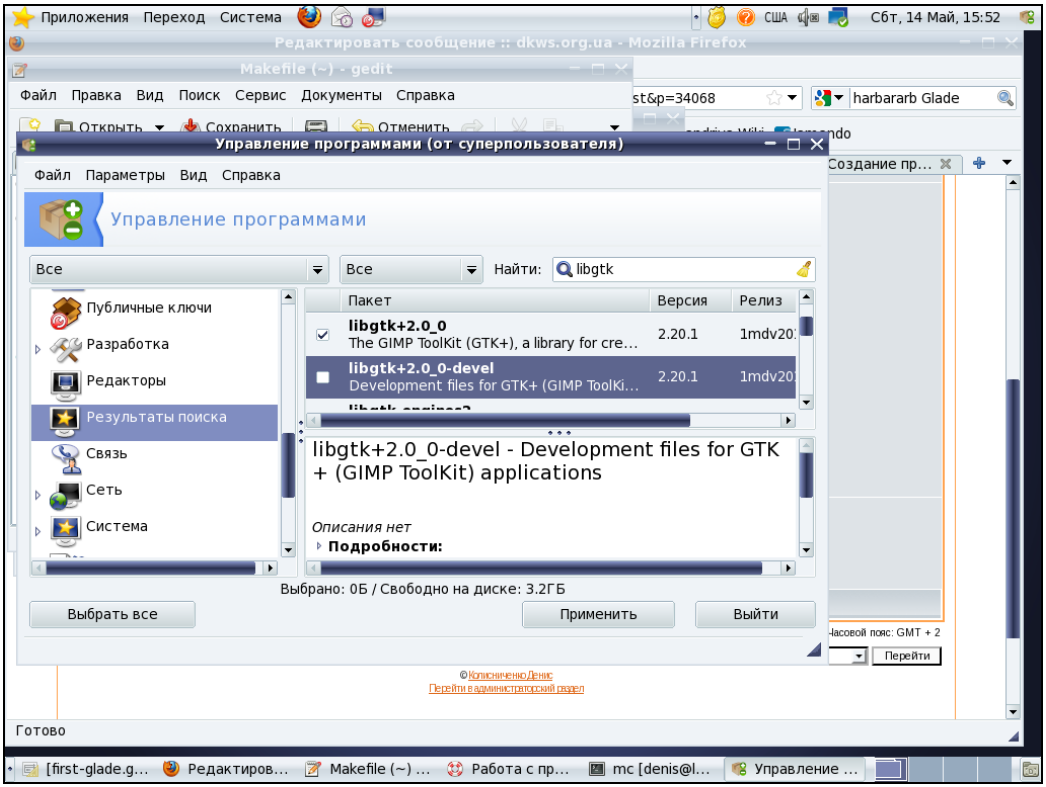


Рис. 33.1. Установка пакета libgtk+2.0_0-devel

glade. Произведите поиск по строке glade в менеджере пакетов и вы найдете нужный пакет.

Кроме этого, если вы планируете программировать на C++, что нередко при создании программ с графическим интерфейсом, вам нужно установить пакеты libgtk+2.0._0-devel (или libgtk2.0-dev) и gcc-c++ (компилятор C++) — рис. 33.1.

33.3. Использование Glade

Использовать Glade довольно просто. Создаете окно, добавляете в него нужный контейнер и размещаете в нем виджеты. После того как виджет размещен в контейнере, вы можете изменять его свойства, например изменить название метки (надписи), ширину, высоту виджета и т. д.

Сейчас мы напишем простейшую программу, которая будет формировать окно с одной кнопкой, при нажатии которой происходит завершение программы. Но на примере этой простой программы будет продемонстрировано все сказанное ранее: размещение виджета, изменение его свойств и установка обработчиков сигналов.

Установите и запустите Glade. При запуске Glade попросит установить параметры проекта (рис. 33.2). Далее вы увидите основное окно Glade. Слева находится панель виджетов, по центру — рабочая область (сразу после создания проекта она будет

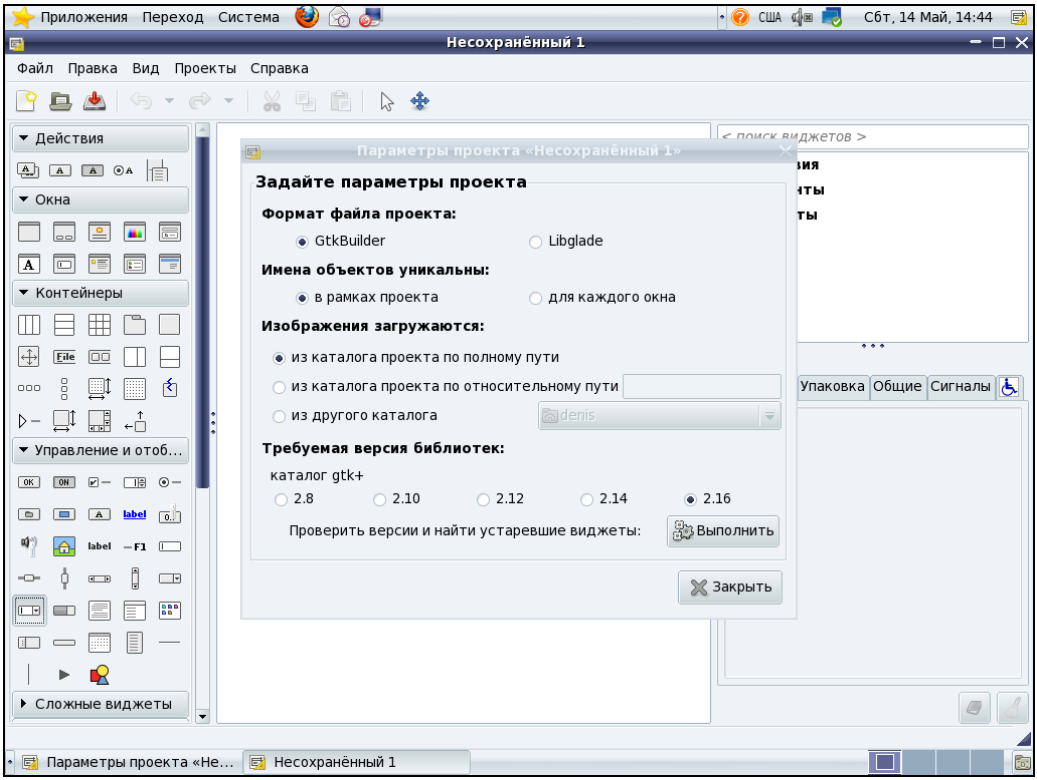


Рис. 33.2. Установка параметров проекта

пустой), слева — размещены две панели: вверху дерево виджетов вашей программы, внизу — панель установки свойств и обработчиков сигнала виджета.

Из области слева выберите окно и поместите его в рабочую область (сначала нужно щелкнуть на изображении окна, затем — на изображении рабочей области) — рис. 33.3.

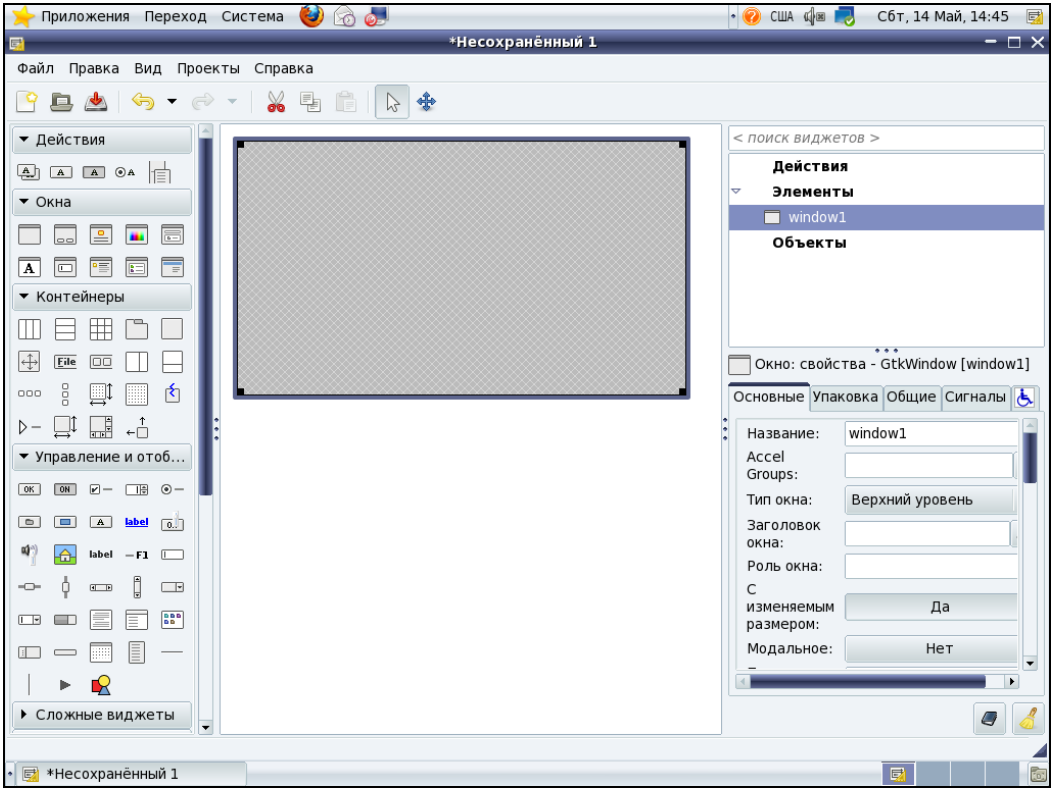


Рис. 33.3. В проект добавлено окно

Затем поместите в окно контейнер. Можно выбрать любой — таблицу, вертикальный контейнер, горизонтальный контейнер, меню — все это зависит от интерфейса вашей программы. Если вы выбрали таблицу, вам будет предложено установить ее размер (рис. 33.4).

После этого выберите кнопку (виджет `button`) и поместите ее в контейнер (щелкните на изображении кнопки и щелкните на нужной ячейке таблицы). После этого посмотрите на дерево виджетов справа: в нем сейчас будут все три использованные вами виджета — окно (`window1`), таблица (`table1`) и кнопка (`button1`). Под деревом виджетов отображаются вкладки, где можно установить свойства виджета и обработчики сигналов. Изменить метку кнопки можно на вкладке **Основные** (рис. 33.5). Редактор Glade сразу поддерживает русский язык, поэтому для корректного отображения кириллицы не нужно производить каких-либо действий.

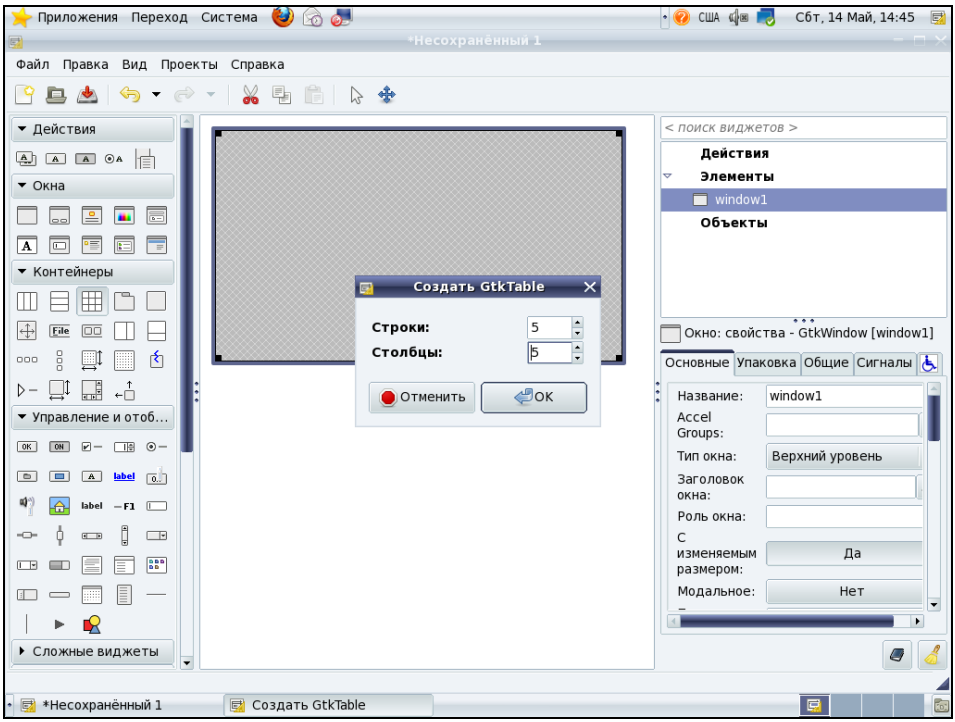


Рис. 33.4. Установка размера таблицы

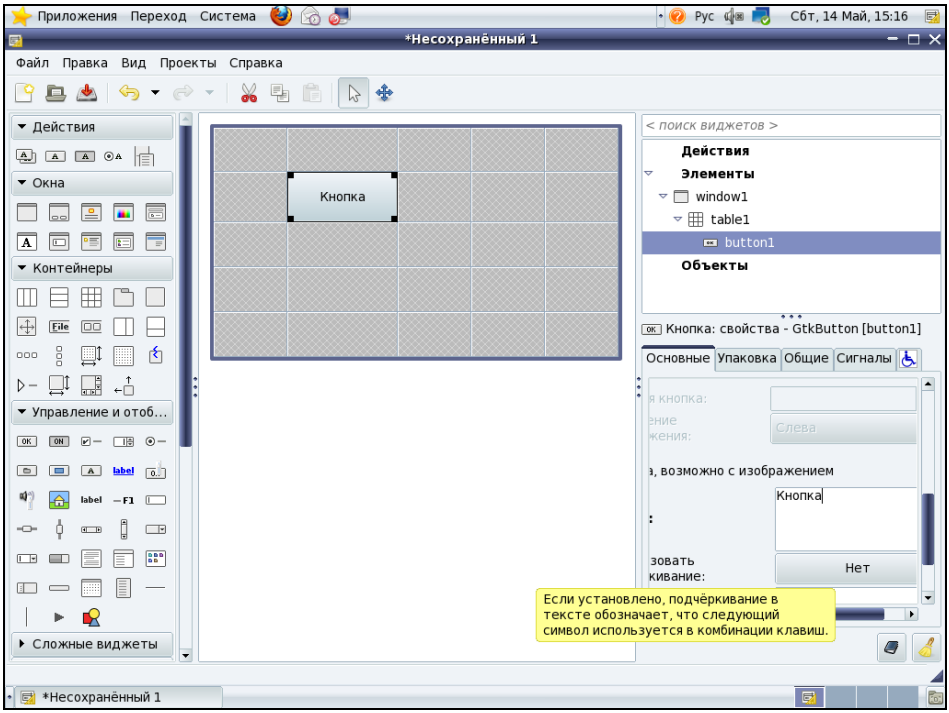


Рис. 33.5. Свойства виджета `button`

Теперь установим обработчик сигнала `clicked` для кнопки. Перейдите на вкладку **Сигналы** и из выпадающего списка выберите обработчик для сигнала, по умолчанию он называется `on_button1_clicked`, но вы можете указать свое название (рис. 33.6). Поскольку наша кнопка будет закрывать программу, то гораздо проще было бы выбрать функцию `gtk_main_quit()`, но это неинтересно. Ведь реальная программа будет выполнять какие-то действия, поэтому мы создадим собственный обработчик, который будет просто вызывать `gtk_main_quit()` — зато вы будете знать, как создаются обработчики сигналов. Если подразумевается наличие кнопки **Выход**, вы можете сразу выбрать обработчик `gtk_main_quit()`.

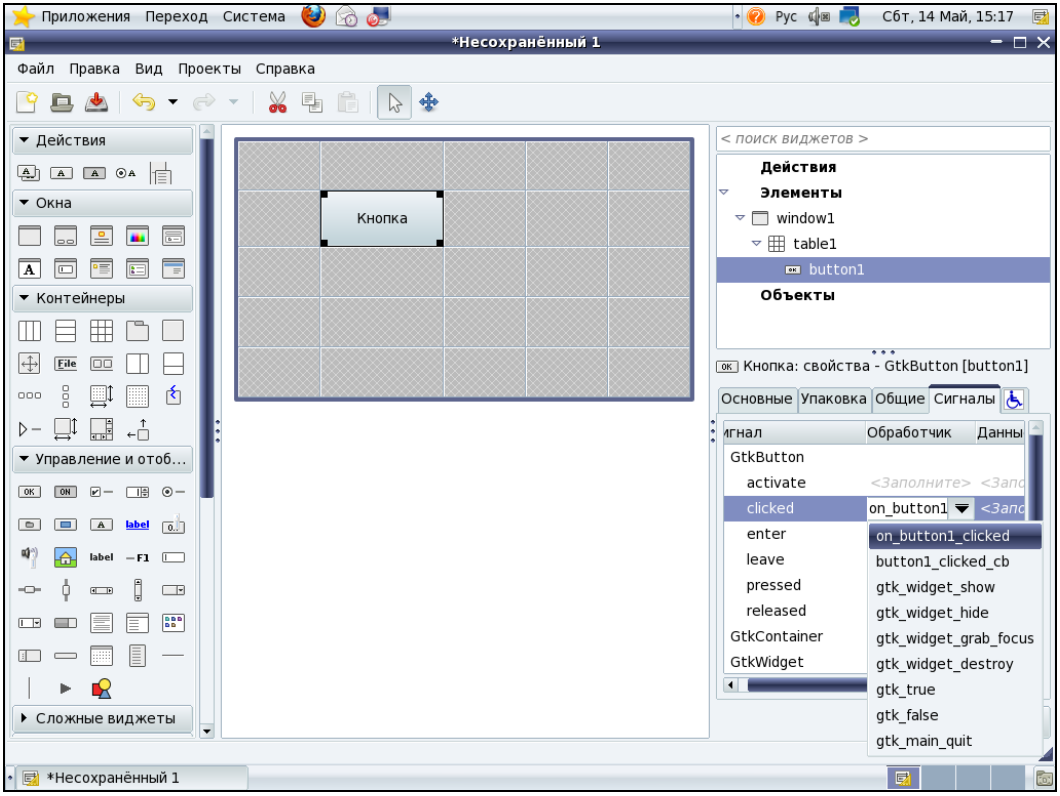


Рис. 33.6. Установка обработчика сигнала `clicked` для кнопки

Не забудьте установить обработчик сигнала `destroy` для основного окна, иначе оно не будет реагировать на нажатие кнопки закрытия окна (точнее, окно-то закроется, но процесс будет по-прежнему "висеть" в памяти). Для этого выберите окно `window1`, перейдите на вкладку **Сигналы** и установите обработчик `gtk_main_quit()` для сигнала `destroy` (рис. 33.7).

Сохраните проект как first-glade — это же имя будет использоваться в листинге 33.1, поэтому, если вы указали другое название, вам придется изменить код программы.

Вот, по большому счету, и все. Думаю, основной принцип вы уловили. Перейдем к написанию программы.

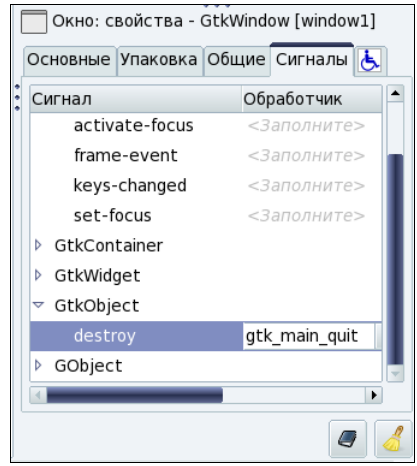


Рис. 33.7. Установка обработчика сигнала `destroy` для окна

33.4. Создание программы

Создать программу можно как на языке C, так и на языке C++ (разница будет, конечно, в синтаксисе и в вызываемом компиляторе, но это уже при сборке программы).

В этой главе напишем программу на C++, дабы никто не посмел упрекнуть в том, что этому языку в книге не уделяется внимание. В листинге 33.1 приводится каркас программы, предназначенной для использования файлов интерфейсов, построенных с помощью Glade.

Листинг 33.1. Каркас программы first-glade.c

```
#include <gtk/gtk.h>
// Обратите внимание: locale.h подключать не нужно, как и вызывать
// setlocale для установки локали — с русскими буквами все будет
// в порядке

// Указываем файл проекта Glade
#define UI_FILE "first-glade.glade"

// Описываем виджеты программы
GtkBuilder *builder;          // "строитель" интерфейса, обязательно
GtkWidget *window1;           // Окно
GtkWidget *button1;           // Кнопка

// Описываем обработчики сигналов
// Пока есть только один обработчик — нажатия единственной кнопки
extern "C" void on_button1_clicked (GtkObject *object,
                                    gpointer user_data);
```

```
// Название обработчика должно совпадать с указанным в проекте!

int main( int argc, char **argv )
{
    GError *error = NULL;

    // Инициализируем GTK+
    gtk_init( &argc, &argv );

   // Создаем новый объект GtkBuilder
    builder = gtk_builder_new();

   // Загружаем интерфейс из файла проекта
   if( ! gtk_builder_add_from_file( builder, UI_FILE, &error ) )
    {
        g_warning( "%s", error->message );
        g_free( error );
        return( 1 );
    }

   // Связываем виджеты с описанием из файла проекта
   // Имя переменной может быть произвольным, однако в качестве
   // аргумента для gtk_builder_get_object нужно указать имя
   // виджета, указанное в проекте Glade
   window1 = GTK_WIDGET(gtk_builder_get_object(builder, "window1"));
   button1 = GTK_WIDGET(gtk_builder_get_object(builder, "button1"));

   // Привязка сигналов
   gtk_builder_connect_signals (builder, NULL);

   // Освобождение памяти
   g_object_unref( G_OBJECT( builder ) );

   // Показываем окно с виджетами
   gtk_widget_show(window1);

   // Запуск приложения
   gtk_main();

   return( 0 );
}

void on_button1_clicked (GtkObject *object, gpointer user_data)
{
 // Завершаем работу GTK-программы
 gtk_main_quit();
}
```

Как видите, при использовании Glade исходный код существенно сокращается, и вы можете заняться программированием, а не размещением виджетов. Особенно сокращение размера кода будет заметно, когда у вас будет большое окно с множеством виджетов.

33.5. Компиляция программы

Как и в случае с обычной GTK-программой, компилятору нужно передать множество опций, поэтому в этот раз мы создадим Makefile для упрощения процесса сборки. Makefile для сборки нашего проекта first-glade представлен в листинге 33.2.

Листинг 33.2. Makefile для сборки проекта

```
CC=g++
LDLIBS=`pkg-config --libs gtk+-2.0 gmodule-2.0`
CFLAGS=-Wall -g `pkg-config --cflags gtk+-2.0 gmodule-2.0`

first-glade: first-glade.o
	$(CC) $(LDLIBS) first-glade.o -o first-glade

first-glade.o: first-glade.c
	$(CC) $(CFLAGS) -c first-glade.c

clean:
	rm -f first-glade
	rm -f *.o
```

Теперь для компиляции программы достаточно ввести команду:

```
make
```

33.6. Рекомендуемая литература

На мой взгляд, приведенной в книге информации вполне достаточно, чтобы начать программировать с использованием GTK+. Однако вам все равно не обойтись без документации, где описаны все свойства и сигналы виджетов, где представлены все функции библиотеки GTK.

Если вы заинтересовались, вашей "настольной" книгой станет руководство разработчика GTK, доступное по адресу:

http://developer.gnome.org/gtk/

В данном руководстве описаны последние версии библиотеки GTK — 2.24 и 2.22, но на английском языке. Понятно, что родная рубашка ближе к телу, поэтому привожу ссылку на русскоязычное руководство по GTK:

http://www.opennet.ru/docs/RUS/gtk-reference/

Однако в этом руководстве рассматривается версия 2.10, но на первое время этого вам будет достаточно.

Также не забудьте посетить сайт **http://www.gtk.org/**, где вы найдете документацию по GLib, GTK, GDK, а также сможете скачать GTK для Windows и OSX.

Неплохая статья по использованию Glade (если не разберетесь сами) находится по адресу:

http://habrahabr.ru/blogs/development/107403/

Не забывайте о справочной системе Glade — она на русском языке, вызвать ее можно, нажав <F1> ☺

Вот теперь можно перейти к следующей части книги, где будет рассмотрена отладка и оптимизация программы.

ЧАСТЬ VIII

Отладка и оптимизация программы

Глава 34.	Отладка программ. Трассировка системных вызовов
Глава 35.	Оптимизация программы

Последняя часть книги посвящена отладке и оптимизации программы средствами `gdb` и `gprof`.

ГЛАВА 34

Отладка программ. Трассировка системных вызовов

34.1. Для чего нужна отладка программ

Когда человек учится программировать, то бóльшая часть ошибок в коде программы — синтаксические. Со временем синтаксических ошибок становится все меньше и меньше. Но синтаксические ошибки — не самое страшное, ведь их определяет компилятор и они устраняются сразу же. После исправления синтаксических ошибок программа будет работать.

Но как она будет работать? Ошибок вроде бы нет, программа работает, но работает она не так, как мы от нее ожидаем: или неправильно обрабатывает данные, или вообще "вылетает" с ошибкой времени выполнения (runtime error). Проблема может заключаться или в неправильном алгоритме, или в его неправильной реализации. Всякое бывает: вы нашли алгоритм (или же разработали его самостоятельно), но или неправильно его реализовали, или просто что-то не учли. Однако с синтаксической точки зрения все нормально — ошибок нет. Компилятор не сообщает о так называемых логических ошибках в программе. Все просто: пока еще не разработали компилятор, читающий мысли программиста. Как только будет такой компилятор разработан, не нужен будет и сам программист — компилятор сможет сам писать код программы — вам нужно будет только объяснить, чего вам хочется от него.

В некоторых случаях логические ошибки проявляются не всегда. Бывает так, что программа нормально работает первые несколько месяцев и все довольны — и разработчик и заказчик. Но через некоторое время ошибки дают о себе знать. Причины таких ошибок разные: от недоработок алгоритма до запуска программы на другом оборудовании — все зависит от специфики программы.

Думаю, вы уже догадались, что логические ошибки — самые страшные. Вроде бы все правильно, но программа не работает как нужно. Далее мы рассмотрим несколько логических ошибок, совершаемых начинающими программистами (хотя такую ошибку может допустить по невнимательности и опытный программист). Конечно, все логические ошибки мы рассмотреть не можем физически — ведь для этого нужно проанализировать все алгоритмы в мире.

Самая тривиальная логическая ошибка — неправильное использование операций инкремента и декремента, например следующие выражения не эквивалентны:

```
a = b++ + 10;
a = ++b + 10;
```

В первом случае переменной `x` будет присвоено значение 10 (при условии, что `b = 1`), а во втором — 11. Значение, на 1 превышающее ожидаемое, может стать причиной неправильной работы программы. Честно говоря, данная ошибка является просто незнанием синтаксиса языка программирования, но, тем не менее, она имеет место.

Следующая ошибка может возникнуть по вашей невнимательности — это ошибка неучтенной единицы. В самом простейшем случае, когда эту ошибку действительно легко обнаружить — это когда происходит выход за границы массива. Вы объявляете массив, скажем из 100 элементов, а затем пытаетесь его инициализировать с помощью цикла:

```
for (i=0;i<=100;i++) a[i] = 0;
```

С точки зрения синтаксиса все правильно, но проблема в том, что программа пытается инициализировать элемент массива с индексом 100, в то время как в массиве есть элементы с индексами от 0 до 99.

Некоторые начинающие программисты допускают двойную ошибку — забывают, что нумерация массивов начинается с 0, и пытаются инициализировать элементы с индексами от 1 до 100:

```
for (i=1;i<=100;i++) a[i] = 0;
```

Но, согласитесь, все эти ошибки легко исправляются даже и без отладчика. Но в C/C++ есть особые ошибки, которые очень хорошо замаскированы и без отладчика не обойтись — это ошибки, связанные с указателями. Указатели в C — очень мощный, но и в то же время очень коварный инструмент. Очень легко выделить недостаточное количество памяти для помещения в нее какого-нибудь объекта — такую ошибку могут допустить даже опытные программисты. Начинающие же обычно "страдают" неправильным использованием операторов `*` и `&`, а также забывают инициализировать указатели и сразу начинают их использовать.

После ошибок с указателями следуют ошибки рекурсивных алгоритмов. Рекурсивная функция вызывает саму себя, но с другими параметрами. Но рано или поздно функция должна возвратить какое-то значение, а не опять вызвать саму себя. Если программист забыл (или неправильно указал) условие выхода из рекурсии, функция будет вызывать саму себя снова и снова, пока не переполнит стек.

Для облегчения поиска ошибок используются отладчики. В Linux имеется свой "придворный" отладчик `gdb` (The GNU Debugger), который и будет рассмотрен в этой книге. Отладчик `gdb` является довольно мощным инструментов для поиска всякого рода логических ошибок и входит в состав практически всех дистрибутивов Linux (во всяком случае, мне не встречались дистрибутивы, в которых бы не было `gdb`).

При отладке программы используют следующие приемы:

- *пошаговое выполнение программы* — программа выполняется оператор за оператором, но перед выполнением следующего оператора у вас есть возможность просмотреть содержимое интересующих вас переменных. Такой прием подходит или для отладки небольших программ (согласитесь, когда в вашей программе несколько тысяч строк, пошаговое выполнение — не самое лучшее решение), или в совсем серьезных случаях, когда другие методы не помогают;
- *установка точек останова* (breakpoint) — программа будет выполняться до установленной точки останова. Затем выполнение программы будет приостановлено, а вы сможете просмотреть содержимое переменных. Преимущество перед предыдущим способом заключается в том, что вам не нужно пошагово выполнять программу до вероятного места ошибки. Вы можете установить несколько точек останова, что делает процесс отладки еще более удобным. После приостановки программы вы можете или продолжить ее выполнение, или завершить ее работу;
- *установка условия останова программы* — можете установить условие, когда программа должна быть остановлена.

Самое время перейти к следующему разделу, в котором мы познакомимся с отладчиком `gdb`.

34.2. Введение в отладчик *gdb*

У отладчика `gdb` довольно много параметров, все эти параметры представлены в табл. 34.1. Формат вызова отладчика `gdb` следующий:

```
gdb [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev] [-s symfile]
[-e prog] [-se prog] [-c core] [-x cmds] [-d dir] [prog[core|procID]]
```

Таблица 34.1. *Параметры* `gdb`

Параметр	Описание
`-help` или `-h`	Выводит подсказку — краткое описание всех параметров
`-nx` или `-n`	Запрещает обрабатывать команды файла инициализации .gdbinit
`-q`	Запрещает выводить приветствие и информацию об авторских правах
`-batch`	Командный режим. Отладчик возвращает 0, если были выполнены все команды, указанные в файле, заданном параметром `-x` (и файле .gdbinit, если его использование разрешено). В противном случае возвращается ненулевое значение
`-cd=каталог`	Используется для установки рабочего каталога (по умолчанию используется текущий каталог)

Таблица 34.1 *(окончание)*

Параметр	Описание
`-f` или `-fullname`	Параметр нужен, если вы планируете использовать интерфейс текстового процессора Emacs для отладки ваших программ с помощью `gdb`. Подробное описание этого параметра вы найдете в справочной системе. На практике Emacs используется редко — преимущественно теми программистами, которые давным-давно привыкли к такому способу отладки
`-b bps (bits per second)`	Используется при удаленной отладке программы. Позволяет установить скорость работы последовательного интерфейса
`-tty=терминал`	Позволяет установить `терминал` в качестве стандартного ввода и вывода для отлаживаемой программы. Довольно удобный параметр: отлаживать программу будете на одном терминале, а смотреть ее вывод — на другом
`-s файл` или `-symbols=файл`	Читает таблицу символов из указанного файла
`-write`	Разрешает запись в исполнимые и core-файлы
`-e программа`	Указанная параметром программа используется как фильтр дампа
`-se=файл`	Читать таблицу символов из указанного файла и использовать указанный файл в качестве исполнимого
`-core=файл` или `-c файл`	Задает core-файл (содержит дамп)
`-command=файл` или `-x файл`	Задает файл, содержащий команды, которые нужно выполнить отладчику (используется в командном режиме)
`-d каталог`	Добавляет каталог к списку поиска исходных текстов
`[prog|core|procID]`	Задает программу для отладки. Вы можете указать исполнимый файл программы (`prog`), дамп-файл (`core`) или же номер процесса (`procID`) — тогда отладчик подсоединится к уже запущенному процессу

Отладка программы возможна только в том случае, если она скомпилирована с опцией `-g`, которая помещает отладочную информацию в исполнимый файл:

```
gcc -g -o prog prog.c
```

Затем можно запустить отладчик так:

```
gdb prog
```

Если же во время выполнения программы произошла ошибка и был создан дамп-файл (core), то укажите и его:

```
gdb prog core
```

Для подключения к уже запущенному процессу нужно указать PID процесса:

```
gdb 1234
```

Отладка программы происходит в интерактивном режиме: вы вводите команду — отладчик ее выполняет. Часто используемые команды отладчика представлены в табл. 34.2.

Таблица 34.2. *Команды отладчика*

Команда	Описание
`break [файл:]функция` `break [файл:]строка`	Устанавливает точку останова. Точку останова не обязательно устанавливать на функции, вы можете указать номер строки, в которой нужно установить точку останова, или же указать имя файла и номер строки в этом файле, если у вас сложная программа, состоящая из нескольких файлов
`run [аргументы]`	Запускает программу и передает ей указанные аргументы
`bt`	Обратная трассировка: отобразить стек программы
`print выражение`	Вывести значение выражения, операндами могут быть переменные, объявленные в вашей программе. Для вывода значения переменной можно просто указать ее имя
`c`	Продолжить выполнение программы (после останова)
`kill`	Завершить программу и процесс отладки
`list`	Показывает исходный код программы
`next`	Выполнить следующую строку. Это один из вариантов пошагового выполнения программы. Если следующим "шагом" будет вызов функции, то отладчик выполнит ее за один шаг
`step`	Второй вариант пошагового выполнения. Отличается от `next` тем, что если будет встречен вызов функции, то каждый оператор функции будет выполнен как отдельный шаг, что усложняет отладку, зато позволяет отлаживать и функции. Отлаживать вы можете только собственные функции, вы не можете произвести отладку функций C, поскольку библиотеки C не собраны с опцией `-g`
`help [имя]`	Вывести справку о команде отладчика или список разделов справочной системы отладчика
`quit`	Завершает работу отладчика
`watch [переменная]`	"Мониторинг" переменной: как только изменится значение указанной переменной, отладчик сразу же сообщит вам об этом. Команда `watch` используется в комбинации с командами `next`/`step`, команда `watch` не приостанавливает выполнение программы при изменении переменной. Намного удобнее для отладки программы использовать команду `awatch`
`awatch [переменная]`	Приостанавливает программу всякий раз, когда производится доступ (чтение или запись) к указанной переменной. Команда `awatch` намного удобнее команды `watch`

Понятно, что в табл. 34.2 представлены не все команды отладчика, но, как правило, этих команд достаточно для отладки программ. Получить информацию о других командах можно с помощью самого отладчика — просто введите команду `help`.

34.3. Пример использования *gdb*

Сейчас мы рассмотрим пример отладки программы с помощью `gdb`. Наша простая программа будет порождать ошибку сегментации. Код программы очень и очень прост — он демонстрирует ошибочное использование оператора `*` в C (листинг 34.1).

Листинг 34.1. Код программы error.c

```
#include <stdio.h>

int main()
{
        char *name = "Denis";

        *name = 'A';

        return 0;
}
```

Посмотрите на рис. 34.1: сначала я вызвал редактор `nano` для создания файла error.c, затем скомпилировал программу (с опцией `-g`) и запустил ее. Компилятор об ошибке не сообщил — программа скомпилирована без ошибок, но при запуске программы я получил ошибку сегментирования.

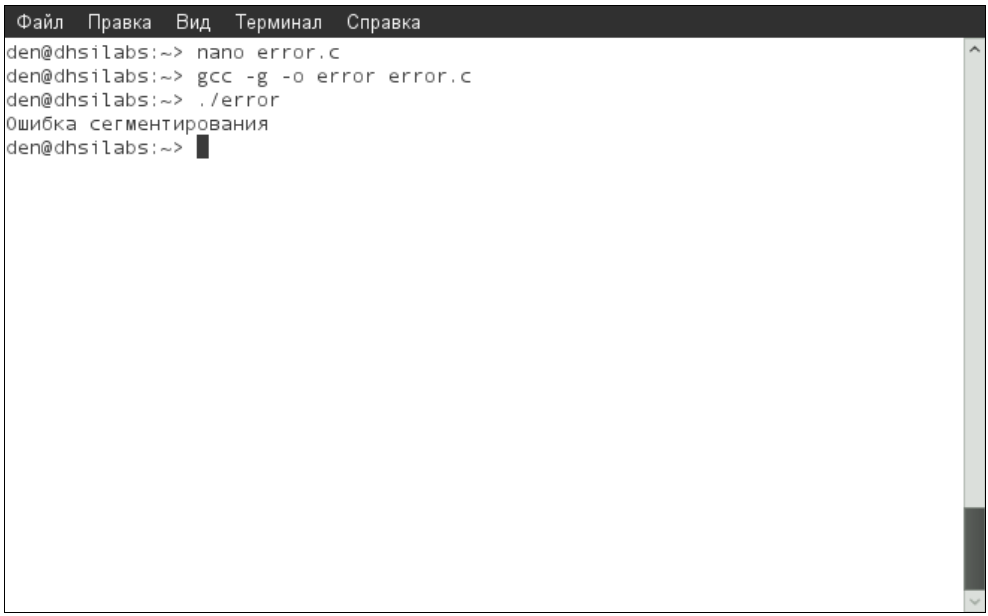


Рис. 34.1. Компиляция и запуск программы с ошибкой

Запустим отладчик:

```
gdb error
```

Примечание

Программе можно передать параметры, например `gdb error option1 option2`.

Отладчик загружает программу, но пока не запускает ее (рис. 34.2).

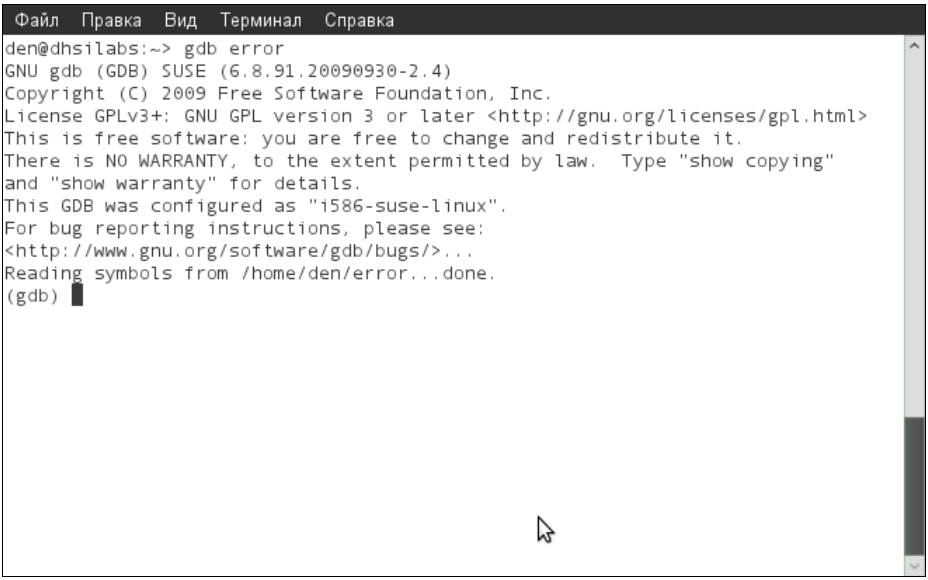


Рис. 34.2. Программа загружена, но не запущена

Для запуска программы нужно ввести команду `run`. Мы сразу же получим сообщение об ошибке сегментации, отладчик также сообщит номер строки, которая вызвала ошибку (рис. 34.3).

Наш пример слишком тривиален — отладчик сразу же показал ошибку и номер строки, в которой она возникает, но, тем не менее, попробуем его немного усложнить. Перезапустите отладчик снова. Перед запуском программы лучше установить точку останова:

```
break main
```

Мы установим точку останова на функции `main` — как только отладчик достигнет этой функции, выполнение программы будет приостановлено. По сути, т. к. `main` — основная функция, то выполнение будет приостановлено сразу же после запуска программы.

Примечание

Если вы хотите продолжить нормальное выполнение (а не пошаговое) программы после точки останова, используйте команду `c`.

Затем нужно ввести `run` — начнется выполнение программы, потом оно будет приостановлено на первой точке доступа (рис. 34.4). Если в данный момент попытаться

```
Файл  Правка  Вид  Терминал  Справка
den@dhsilabs:~> gdb error
GNU gdb (GDB) SUSE (6.8.91.20090930-2.4)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i586-suse-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/den/error...done.
(gdb) run
Starting program: /home/den/error
Missing separate debuginfo for /lib/ld-linux.so.2
Try: zypper install -C "debuginfo(build-id)=d7706cbaa0ca09319cb645eac789cb839907
8797"
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=ee302691046515fe3766ae3b7d47afd3e3a8
d063"

Program received signal SIGSEGV, Segmentation fault.
0x080483f4 in main () at error.c:7
7       *name = 'A';
(gdb)
```

Рис. 34.3. Ошибка сегментации

```
Файл  Правка  Вид  Терминал  Справка
den@dhsilabs:~> clear

den@dhsilabs:~> gdb error
GNU gdb (GDB) SUSE (6.8.91.20090930-2.4)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i586-suse-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/den/error...done.
(gdb) break main
Breakpoint 1 at 0x80483ea: file error.c, line 5.
(gdb) run
Starting program: /home/den/error
Missing separate debuginfo for /lib/ld-linux.so.2
Try: zypper install -C "debuginfo(build-id)=d7706cbaa0ca09319cb645eac789cb839907
8797"
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=ee302691046515fe3766ae3b7d47afd3e3a8
d063"

Breakpoint 1, main () at error.c:5
5       char *name = "Denis";
(gdb) print name
$1 = 0x0
(gdb)
```

Рис. 34.4. Установка точки останова и запуск программы

```
Файл  Правка  Вид  Терминал  Справка
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/den/error...done.
(gdb) break main
Breakpoint 1 at 0x80483ea: file error.c, line 5.
(gdb) run
Starting program: /home/den/error
Missing separate debuginfo for /lib/ld-linux.so.2
Try: zypper install -C "debuginfo(build-id)=d7706cbaa0ca09319cb645eac789cb839907
8797"
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=ee302691046515fe3766ae3b7d47afd3e3a8
d063"

Breakpoint 1, main () at error.c:5
5       char *name = "Denis";
(gdb) print name
$1 = 0x0
(gdb) awatch name
Hardware access (read/write) watchpoint 2: name
(gdb) step
7       *name = 'A';
(gdb) step
Hardware access (read/write) watchpoint 2: name

Old value = 0x0
New value = 0x80484d0 "Denis"
0x080483f4 in main () at error.c:7
7       *name = 'A';
(gdb)
```

Рис. 34.5. Команда `awatch` в действии

```
Файл  Правка  Вид  Терминал  Справка
Starting program: /home/den/error
Missing separate debuginfo for /lib/ld-linux.so.2
Try: zypper install -C "debuginfo(build-id)=d7706cbaa0ca09319cb645eac789cb839907
8797"
Missing separate debuginfo for /lib/libc.so.6
Try: zypper install -C "debuginfo(build-id)=ee302691046515fe3766ae3b7d47afd3e3a8
d063"

Breakpoint 1, main () at error.c:5
5       char *name = "Denis";
(gdb) print name
$1 = 0x0
(gdb) awatch name
Hardware access (read/write) watchpoint 2: name
(gdb) step
7       *name = 'A';
(gdb) step
Hardware access (read/write) watchpoint 2: name

Old value = 0x0
New value = 0x80484d0 "Denis"
0x080483f4 in main () at error.c:7
7       *name = 'A';
(gdb) step

Program received signal SIGSEGV, Segmentation fault.
0x080483f4 in main () at error.c:7
7       *name = 'A';
(gdb)
```

Рис. 34.6. Ошибка сегментации после команды `step`

вывести значение переменной `name`, то мы ничего полезного не увидим, поскольку `name` еще не объявлена.

Теперь посмотрите на рис. 34.5. Я установил наблюдение (с помощью команды `awatch`) за переменной `name` и использовал команду `step` для пошагового выполнения программы. Команда `awatch` приостанавливала выполнение программы как только изменялось значение переменной `name`. После очередной паузы, вызванной командой `awatch`, я снова ввел `step` и получил ошибку сегментации (рис. 34.6).

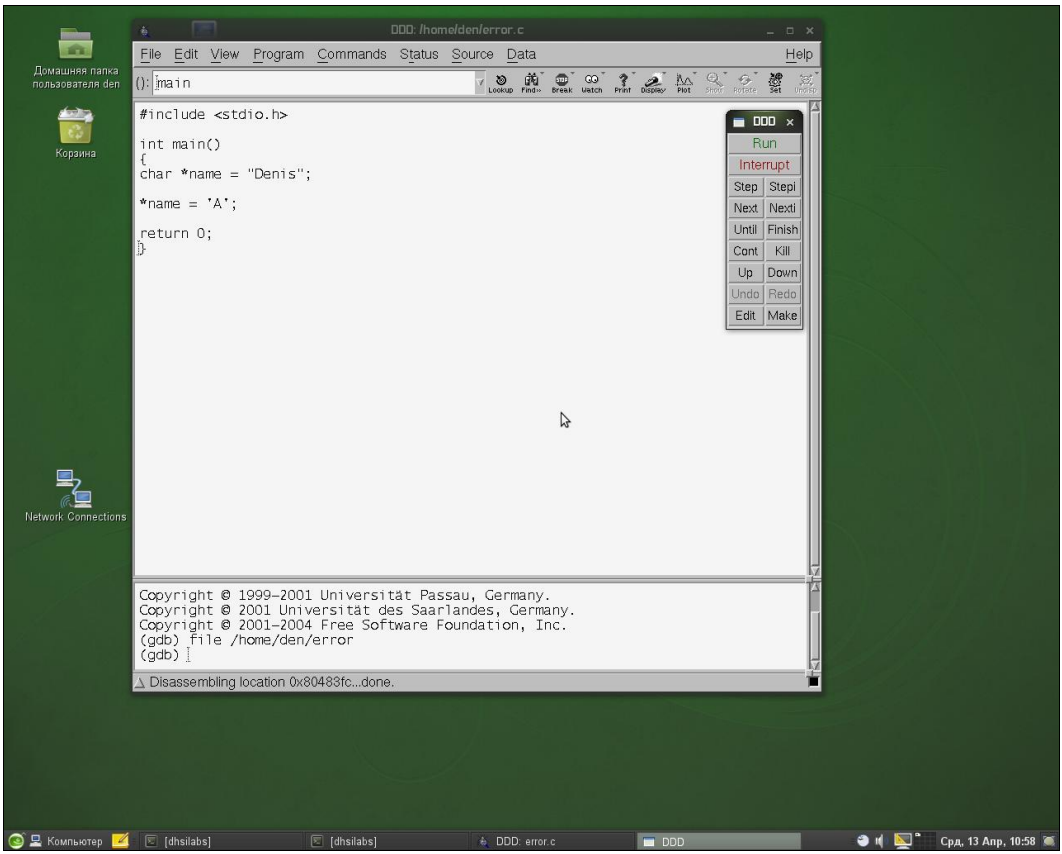


Рис. 34.7. Оболочка DDD

Некоторым пользователям удобнее использовать графическую оболочку для `gdb`, чем консоль и интерактивный режим `gdb`. Для отладчика `gdb` было разработано много разных оболочек, одна из самых удачных называется DDD. Разобраться с оболочкой довольно просто, поэтому подробно мы ее рассматривать не будем (рис. 34.7).

34.4. Трассировка системных вызовов

Не знаю, как вам, а мне, сугубо в образовательных целях, всегда было интересно, какие системные вызовы использует моя программа. Удовлетворить любопытство помогла программа `strace`, позволяющая проследить, какие системные вызовы использует программа. Оказывается, даже самая простая программа порождает много системных вызовов, в чем мы скоро убедимся.

Формат вызова `strace` следующий:

```
strace [ -dffhiqrtttTvxx ] [ -acolumn ] [ -eexpr ] ... [ -ofile ] [ -ppid ] ...
[ -sstrsize ] [ -uusername ] [ command [ arg ... ] ]
```

Рассмотреть параметры `strace` мы всегда успеем. Давайте проследим системные вызовы какой-нибудь программы. Возьмем самую простую программу — пусть ей будет наша программа с ошибкой из предыдущего раздела:

```
strace error
```

Вы только посмотрите на рис. 34.8 — вы, наверное, и не ожидали такого количества системных вызовов от одной маленькой программы?

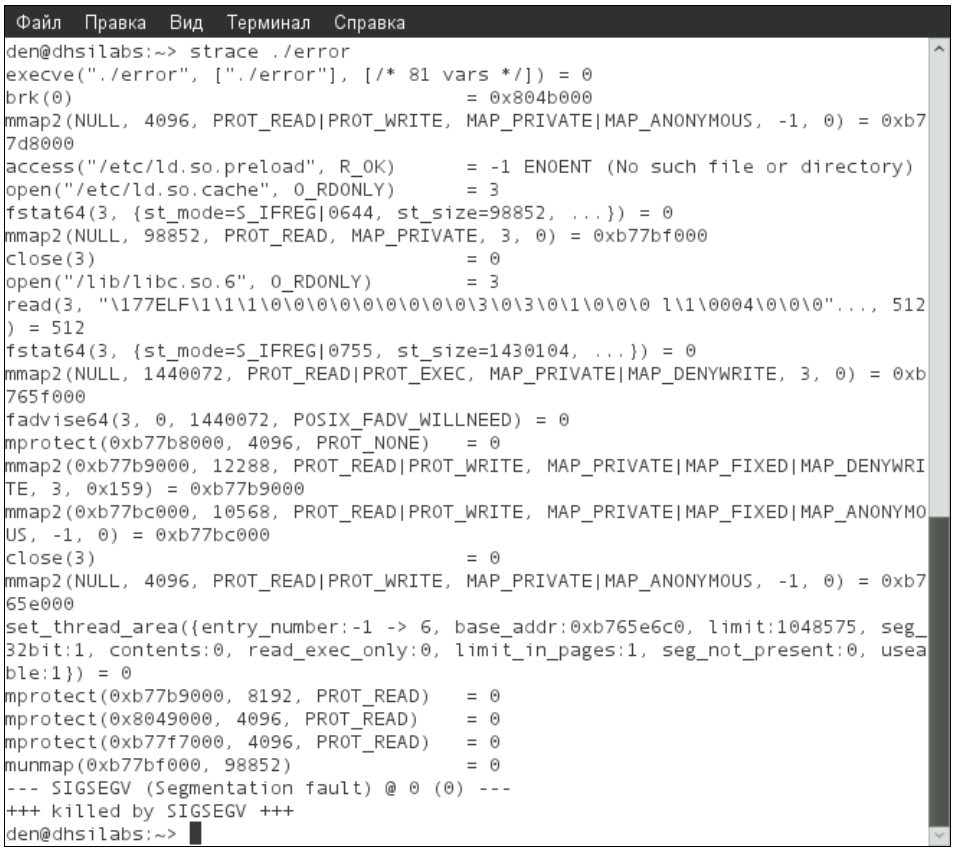


Рис. 34.8. Вывод `strace`

Формат вывода strace следующий:

```
системный вызов = возвращаемое значение
```

Например:

```
open("/lib/libc.so.6/ O_RDONLY) = 3
```

Вот пример обозначения сигналов:

```
--- SIGINT (Interrupt) ---
+++ killed by SIGINT +++
```

Правда, в случае с нашей программой процесс "убивается" сигналом SIGSEGV (Segmentation fault).

Список параметров strace для более подробной трассировки представлен в табл. 34.3.

Таблица 34.3. *Параметры программы* `strace`

Параметр	Описание
`-c`	Подсчитывает время, затраченное на каждый вызов и обработку ошибок. В конце трассировки будет представлен подробный отчет
`-d`	Вывести отладочные сообщения самой программы `strace` на стандартный вывод ошибки
`-f`	Позволяет трассировать дочерние процессы, созданные уже трассируемыми процессами
`-ff`	Записывает каждый трассируемый процесс в файл имя_файла.pid. Данный параметр применятся только вместе с параметром `-o имя_файла`
`-F`	Следовать вызовам `vfork()`. Данный параметр нельзя использовать вместе с опцией `-f`
`-h`	Отображает справку
`-i`	Выводит указатель инструкции во время системного вызова
`-q`	Подавляет вывод некоторых сообщений
`-r`	Выводит относительный штамп времени для каждого вызова
`-t`	Перед каждой строкой будет выводиться текущее время
`-tt`	То же, что и `-t`, но будут выводиться также микросекунды
`-T`	Показывает время, потраченное на каждый системный вызов (т. е. разницу между временем запуска и временем завершения вызова)
`-v`	Выводит дополнительную информацию
`-V`	Вывести номер версии `strace`
`-x`	Строки в кодировке не-ASCII будут выводиться в шестнадцатеричном формате
`-xx`	Выводить все строки в шестнадцатеричном формате (не очень удобный режим!)

Таблица 34.3 *(окончание)*

Параметр	Описание
`-a столбец`	Позволяет выровнять возвращаемые вызовами значения в указанном столбце (по умолчанию — 40)
`-e trace=набор`	Позволяет определить набор отслеживаемых вызовов. Например, `trace=open,close,read,write`
`-e trace=file`	Отслеживает только системные вызовы для работы с файлами (`open`, `close`, `stat`, `chmod`, `unlink` и т. д.)
`-e trace=process`	Отслеживает вызовы для работы с процессами (`fork`, `exec`, `wait` и др.)
`-e trace=network`	Отслеживает сетевые системные вызовы
`-e trace=signal`	Отслеживает вызовы для работы с сигналами
`-e trace=ipc`	Отслеживает IPC-вызовы (вызовы для межпроцессного взаимодействия)
`-e abbrev=набор`	Позволяет сократить вывод каждого члена структуры. Например, `abbrev=all` или `abbrev=none`
`-e verbose=набор`	Различает структуры различных системных вызовов, по умолчанию `verbose=all`
`-e raw=set`	Выводит недекодированные значения аргументов системных вызовов. Полезно, если вы хотите знать точное представление аргумента
`-e signal=набор`	Позволяет определить набор трассируемых сигналов. По умолчанию `signal=all`. Вы можете использовать восклицательный знак для отрицания, например `signal=!SIGTERM` означает, что сигнал `SIGTERM` не будет трассирован
`-e read=набор`	Используется для полного шестнадцатеричного и ASCII-дампа всех прочитанных вызовом `read()` данных. Например, чтобы видеть все данные, поступающие через дескрипторы 2 и 7, введите `read=2,7`
`-e write=набор`	То же, что и `-e read`, но только для записи
`-o имя_файла`	Перенаправляет вывод программы в указанный файл. Данный файл будет полезен для дальнейшего анализа трассировки
`-p pid`	Присоединиться к процессу с `PID=pid` и начать трассировку
`-s размер`	Устанавливает максимальный размер строки (по умолчанию — 32). Имена файлов не рассматриваются как строки и всегда будут напечатаны полностью
`-S критерий`	Сортирует вывод гистограммы, которая выводится опцией `-c`, по заданному критерию: `time` (время), `calls` (вызовы), `name` (имя) и `nothing` (без сортировки)
`-u имя_пользователя`	Запускает программу от имени указанного пользователя. Позволяет проверить, как будет работать программа от имени конкретного пользователя

ГЛАВА 35

Оптимизация программы

35.1. Назначение и основные опции профайлера *gprof*

Представим, что у нас есть медленно работающая программа. Скорее всего, причина кроется в неэффективном, медленном алгоритме. Конечно, если не принимать во внимание медленный компьютер. Существуют программы, позволяющие определить время работы каждой функции программы и всей программы в целом. Программы такого рода называются профайлерами. Обычно по умолчанию устанавливается профайлер `gprof`. Самое интересное, что в моем дистрибутиве openSUSE по умолчанию был установлен `gprof`, но не был установлен даже компилятор `gcc`. Зачем нужен профайлер без компилятора — ума не приложу.

В этой главе мы рассмотрим программу `gprof` (The GNU Profiler), позволяющую определить время работы каждой функции. Самые полезные опции программы представлены в табл. 35.1.

Таблица 35.1. *Некоторые опции программы* `gprof`

Опция	Описание
`-a`	Не выводить информацию о статических функциях
`-b`	Не выводить описание полей таблицы результатов
`-c`	Включить эвристический анализ текстового сегмента объектного файла с целью создания статического графика вызовов
`-e имя_функции`	Не выводить отчет о работе указанной функции и обо всех функциях, которые из нее вызываются
`-E имя_функции`	Не обрабатывать указанную функцию и все функции, которые она вызывает
`-f имя_функции`	Выводить информацию только об указанной функции и обо всех функциях, которые из нее вызываются
`-F имя_функции`	Обрабатывать только указанную функцию и все функции, которые из нее вызываются
`-k func1 func2`	Не выводить информацию о вызове функции `func2` из функции `func1`

Таблица 35.1 *(окончание)*

Опция	Описание
`-s`	Создание итогового файла gmon.sum
`-T`	Вывод в традиционном BSD-стиле, подойдет для программистов, ранее использовавших профайлер в BSD
`-Q` (или `--no-graph`)	Не выводить диаграмму вызовов функцией, делает вывод более компактным
`-w ширина`	Задает "ширину" вывода в знакоместах
`-z`	Выводить функции с нулевым процессорным временем

Об остальных опциях программы `gprof` вы можете прочитать в справочном руководстве по команде `man gprof`.

35.2. Практическое использование профайлера

Для использования профайлера нужно скомпилировать программу с опцией компилятора `-pg`. Если вы этого не сделаете, при запуске профайлера увидите сообщение:

```
gmon.out: no such file or directory
```

Не нужно использовать опцию `-o`, задающую имя результирующего файла, т. к. профайлер по умолчанию работает с файлом a.out.

После компиляции программы запустите файл a.out, чтобы он создал файл gmon.out. Теперь запустите профайлер с параметром `--nograph`:

```
$ gcc -pg demo.c
$ ./a.out
$ gprof --no-graph
```

Результат вывода профайлера с параметром `--no-graph` (он же `-Q`) изображен на рис. 35.1. По умолчанию профайлер выводит кроме полезной информации еще и описание полей таблицы, поэтому я рекомендую использовать параметр `-b`. С описанием полей таблицы результатов профайлера (табл. 35.2) можно ознакомиться один раз, а затем вызывать профайлер только с параметром `-b`. Результат вывода профайлера с параметром `-b` приведен на рис. 35.2.

Таблица 35.2. *Описание таблицы результатов профайлера*

Поле	Описание
`time`	Время работы функции в процентном соотношении
`cumulative seconds`	Сумма числа секунд времени выполнения этой функции и функций, которые вызываются этой функцией (всех дочерних функций)

Таблица 35.2 *(окончание)*

Поле	Описание
`self seconds`	Число секунд, потраченное на работу этой функции в отдельности
`calls`	Число вызовов
`self ms/calls`	Количество миллисекунд, на протяжении которых функция выполнялась
`total ms/calls`	Количество секунд, на протяжении которых выполнялась функция и все функции, которые вызываются данной функцией
`name`	Имя функции

Дабы вы понимали, что делают функции `function` и `function2`, рассмотрим листинг программы demo.c (листинг 35.1).

```
Файл  Правка  Вид  Терминал  Справка

   [1] function                [3] function2
den@dhsilabs:~> gprof ./a.out --no-graph
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 56.90      0.33     0.33       10    33.00    33.00  function2
 43.10      0.58     0.25       10    25.00    58.00  function

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
den@dhsilabs:~>
```

Рис. 35.1. Вывод `gprof` (без графика вызовов функций)

```
Файл  Правка  Вид  Терминал  Справка
den@dhsilabs:~> gprof -b
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 56.90      0.33     0.33       10    33.00    33.00  function2
 43.10      0.58     0.25       10    25.00    58.00  function


                     Call graph


granularity: each sample hit covers 4 byte(s) for 1.72% of 0.58 seconds

index % time    self  children    called     name
                0.25    0.33      10/10          main [2]
[1]    100.0    0.25    0.33      10         function [1]
                0.33    0.00      10/10          function2 [3]
-----------------------------------------------
                                                <spontaneous>
[2]    100.0    0.00    0.58                 main [2]
                0.25    0.33      10/10          function [1]
-----------------------------------------------
                0.33    0.00      10/10          function [1]
[3]     56.9    0.33    0.00      10         function2 [3]
-----------------------------------------------


Index by function name

   [1] function               [3] function2
den@dhsilabs:~>
```

Рис. 35.2. Вывод `gprof` с параметром `-b`

Листинг 35.1. Программа demo.c

```
#include <stdio.h>

int function2()
{
        int i;
        // Небольшая задержка — пустой цикл, который ничего не делает
        for (i=0; i<9999999; i++) ;

        return 111;
}

int function()
{
        int i;
        double x = 7.7723232323;
        double y = 524543.5454;
```

```
        // Еще одна задержка — в цикле производится
        // деление x на y, толку от этого никакого — код
        // приводится ради примера
        for (i=0; i<9999999; i++) x/y;
        // Вызываем функцию function2() с пустым циклом
        function2();

        return x/y;
}

int main()
{
        int i;
        double k;
        for (i=0; i<10; i++)
        {
                printf("%d\b", i);
                k = function();
        }
return 0;
}
```

Проанализируем вывод профайлера (см. рис. 35.1). Как видно из листинга 35.1, функция `function()` вызывается 10 раз (см. поле `calls`). Эта функция вызывает функцию `function2()`, следовательно, число вызовов этой функции тоже равно 10.

Функция `function()` выполняется 0,25 секунды (self seconds), эта функция еще вызывает функцию `function2()`, поэтому поле `cumulative seconds` равно 0,58 (0,25 + 0,33). Посмотрите на рис. 35.2 график вызовов функции `function()`. Выполнение самой функции занимает 0,25 секунды, выполнение дочерних (children) функций — 0,33 секунды.

Функция `function2()` работает 0,33 секунды, она не вызывает других функций, поле `cumulative seconds` равно 0,33 секунды. В графике вызовов указано только время выполнения самой функции (self), время выполнения дочерних функций (children) равно 0.

Самые внимательные читатели, наверное, заметили, что в графике вызовов ничего нет о функции `printf()`, которая вызывается 10 раз, как и функция `function()`. Это происходит потому, что библиотека времени исполнения libc.so не была собрана с параметром `-pg`, поэтому информация профилирования не собирается для функций из этой библиотеки.

Перед тем как приступить к самому процессу профилирования, нужно отметить, что не следует особо доверять числам, выводимым профайлером, поскольку они могут быть неточными из-за округления. Числа используются не для вычисления точного времени выполнения функции, а только для того, чтобы понять, какая функция выполняется быстрее, а какая — медленнее.

Теперь поговорим о самом профилировании, или оптимизации программы. Запустите профайлер и проанализируйте, какие функции выполняются медленнее, чем остальные. Работать нужно над самыми медленными функциями. Понятно, что процессорное время, занимаемое определенной функцией, зависит от алгоритма ее работы. Возможно, что придется разработать другой алгоритм — более эффективный. Если другого алгоритма нет, тогда попробуйте оптимизировать программу с помощью компилятора — используйте одну из опций -O. Собственно, вот и все, что вам нужно знать об оптимизации программы. Дальше нужна только практика, оптимизация — процесс, уникальный для каждой программы. В одной программе нужно оптимизировать одну функцию, в другой — другую. С помощью профайлера вы можете выяснить, какие функции "тормозят" работу программы.

Заключение

Программирование в Linux — довольно интересный процесс, хотя бы потому, что доступны всевозможные средства разработки приложений — от встроенных языков командных оболочек до полноценных языков программирования. Сам процесс обучения программированию существенно облегчен благодаря продуманной справочной системе: любая команда, любой системный вызов описан в `man`. Вам даже не придется бороздить просторы Интернета: все необходимое будет под рукой — на вашем локальном компьютере. То же самое касается и примеров — в /usr/share можно найти много полезных примеров, особенно касающихся TCL/Tk и GTK.

Могу только отметить, что в книге рассмотрены далеко не все средства программирования в Linux. Например, ничего не сказано о библиотеке `Qt`, позволяющей создавать приложения, оптимизированные под использование в графической среде KDE. Библиотека `Qt` не рассматривалась намеренно: в книге описаны две библиотеки для создания графического интерфейса — рассматривать третью нет особого смысла. Если вы надумаете создавать такую программу, то вам пригодится программа QtDesigner, позволяющая за короткое время создать интерфейс пользователя. Также разрешите порекомендовать одну из лучших, на мой взгляд, книг по Qt:

> Шлее М. Qt4.5. Профессиональное программирование на C++ (**http://bhv.ru/books/book.php?id=186572**)

Однако GTK, Qt и C/C++ — это классика программирования под Linux. Если же хочется чего-то эксклюзивного, рекомендую обратить свое внимание на язык Рефал, информацию о котором (в том числе ссылки на учебники) вы найдете на моем сайте:

http://www.dkws.org.ua/phpbb2/viewtopic.php?t=5005

Также обратите свое внимание на компиляторы `Genie` и `Vala`: у первого Python'oобразный синтаксис, а у второго синтаксис напоминает C#. Почему нельзя использовать обычный Python? Да потому что `Genie` — это настоящий компилятор, в отличие от интерпретатора Python, а это означает, что ваши файлы размером в несколько гигабайтов будут "пропарсены" гораздо быстрее, чем в случае с класси-

ческим интерпретатором. Получить информацию об этих интересных продуктах можно по адресу:

http://bkhome.org/genie/index.html

Поклонникам языка Pascal настоятельно рекомендую компилятор Free Pascal Compiler (FPC) и среду разработки приложений Lazarus, которая очень напоминает Borland Delphi.

Напоследок обратите внимание на среду разработки приложений MonoDevelop, ориентированную на язык C#:

http://monodevelop.com/Download

Заключение на фоне других моих книг получилось довольно большим, поэтому самое время традиционно напомнить о своем сайте, где вы можете связаться со мной и задать вопросы, если таковые появятся в ходе чтения книги:

http://www.dkws.org.ua

ПРИЛОЖЕНИЕ

Ядро Linux

Linux, в отличие от многих других операционных систем, позволяет обычному пользователю проникнуть в святая святых — в собственное ядро. Любой желающий может загрузить исходные коды ядра и скомпилировать ядро операционной системы.

Вообще, перекомпиляция ядра — весьма специфическая операция. Раньше ее нужно было делать довольно часто — практически каждый Linux-пользователь со стажем хотя бы раз в жизни перекомпилировал ядро. Зачем? Например, чтобы включить дополнительные функции. Или, наоборот, выключить поддержку некоторых устройств и некоторые ненужные функции — так ядро окажется компактнее и система будет работать быстрее.

Сейчас я даже не знаю, зачем может понадобиться перекомпиляция ядра. Это настолько редкая операция в наше время, что даже исходные коды ядра перестали поставляться на дистрибутивных дисках. Исключение составляет дистрибутив Mandriva 2010 — один из немногих, на дистрибутивном диске которого до сих пор есть исходники ядра. Вот на его примере мы и будем рассматривать компиляцию ядра — ну не хочется мне "тянуть" исходные коды с **www.kernel.org**. В других дистрибутивах процедура перекомпиляции ядра будет такой же, за исключением того, что исходные тексты ядра нужно будет или загрузить из репозитория дистрибутива, или скачать с **www.kernel.org**.

П1. Установка исходных кодов ядра

Запустите программу `drakrpm` и в группе **Разработка | Ядро** найдите и выберите пакет kernel-source (рис. П1). Программа `drakrpm` предложит вам установить дополнительные пакеты — соглашайтесь (рис. П2).

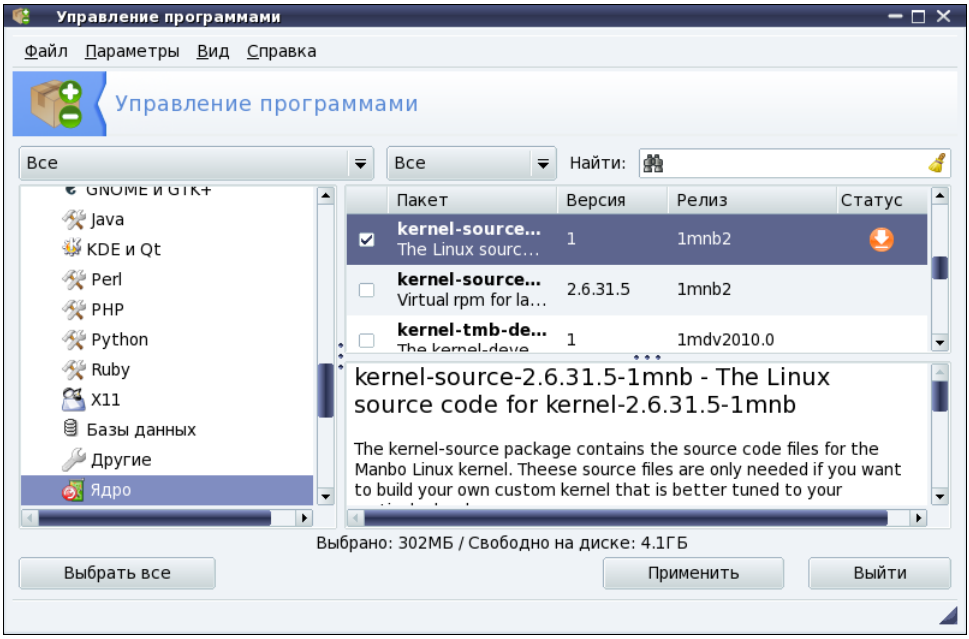


Рис. П1. Установка пакета kernel-source

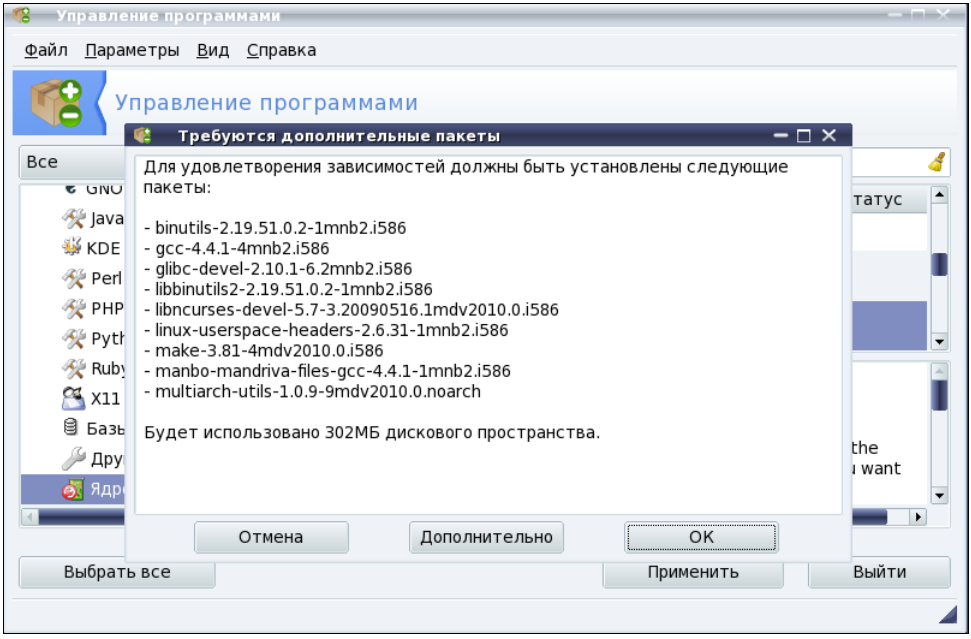


Рис. П2. Установка дополнительных пакетов

П2. Настройка ядра

После установки исходников перейдите в каталог /usr/src/linux:

```
# cd /usr/src/linux
```

Теперь введите одну из двух команд:

```
# make menuconfig
# make xconfig
```

Эти команды предназначены для вызова конфигуратора ядра — с его помощью вы можете включить/выключить функции ядра. Мне больше нравится конфигуратор `menuconfig` (рис. П3) — его можно запускать как в консоли, так и в терминале (если вы предпочитаете графический режим). А `xconfig` можно запускать только в графическом режиме.

> ***Совет***
>
> Графический конфигуратор `xconfig` основан на библиотеке Qt, поэтому он будет работать, если у вас установлена графическая среда KDE. Если вы используете GNOME, то вам нужно запустить конфигуратор `gconfig`, который основан на библиотеке GTK. А еще проще: используйте `menuconfig` — он будет работать в любом случае.

Ядро не компилируют без определенной цели. Скорее всего, вы знаете, какую именно опцию ядра вам нужно включить или, наоборот, выключить. Но если вы хотите скомпилировать ядро эксперимента ради, в качестве путеводителя послужит табл. П1, в которой описаны основные разделы опций ядра.

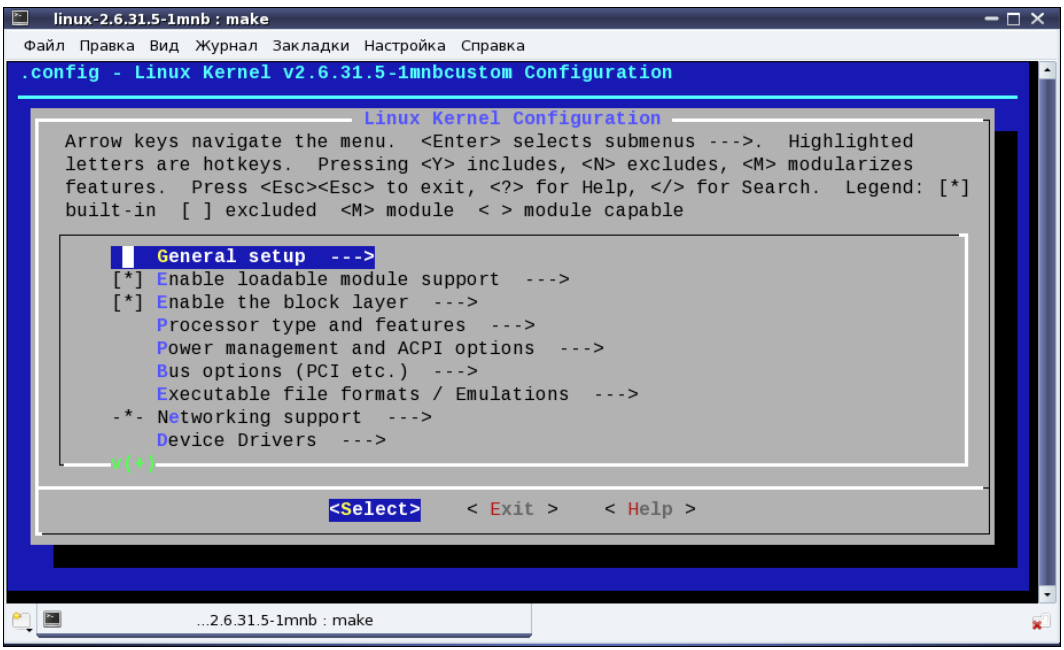


Рис. П3. Конфигуратор `make menuconfig`

Таблица П1. *Разделы опций ядра*

Раздел	Описание
General setup	Общие параметры, например поддержка своп-памяти, межпроцессного взаимодействия System V, Sysctl. Если не знаете, для чего нужна та или иная опция, выделите ее и нажмите клавишу <F1>. А уж если не знаете английский, то до его изучения лучше опции не выключать!
Enable loadable module support	Поддержка загружаемых модулей. Драйверы устройств в Linux разработаны в виде модулей ядра. Здесь вы можете указать, нужна ли вам поддержка модулей. Отключать поддержку модулей на обычных машинах не рекомендуется. Если же вы хотите построить мало обслуживаемый сервер, работающий по принципу "построил и забыл", отключение поддержки загружаемых модулей позволит даже повысить безопасность сервера, поскольку злоумышленник не сможет добавить свой код в ядро путем загрузки модуля. Однако в этом случае ядро будет очень громоздким, потому что вам придется все нужные вам функции, которые были реализованы в виде модулей, компилировать в ядро
Enable the block layer	Включение поддержки больших блочных устройств размером более 2 Тбайт
Processor type and features	Выбор типа процессора и включение/выключение различных функций процессора
Power management and ACPI options	Опции управления питанием (ACPI, APM)
Bus options	Включение/выключение поддержки различных системных шин, а также определение их функции
Executable file formats / Emulations	Параметры поддержки форматов исполнимых файлов
Networking support	Сетевые опции ядра
Device drivers	Драйверы устройств. Здесь вы можете определить, какие устройства система должна поддерживать, а какие — нет
Firmware drivers	Драйверы микропрограммного обеспечения (поддержка различных BIOS)
File systems	Здесь вы можете определить, какие файловые системы должна поддерживать ваша система, а какие — нет
Kernel Hacking	Различные параметры, относящиеся непосредственно к ядру
Security options	Параметры безопасности
Cryptographic API	Параметры криптографии (поддержка различных алгоритмов шифрования данных)
Virtualization	Параметры виртуализации
Library routines	Поддержка различных библиотечных функций (но если заглянуть в этот раздел, то вы увидите, что все эти функции связаны с вычислением контрольной суммы CRC)
Unofficial 3rd party kernel additions	Неофициальные дополнения ядра (от сторонних разработчиков). Сюда разработчики дистрибутива (не ядра) могут включать дополнительные модули. Вообще, будьте осторожны, потому что среди них могут быть экспериментальные функции, включение которых отрицательно отразится на стабильности системы

Опции ядра могут быть либо включены, либо выключены. Если опция выключена, то ее код исключается из ядра (при компиляции ядра он не будет учитываться). Если же опция включена, то ее код будет включен в состав ядра. Но есть еще третье состояние опции — M. Это означает, что опция будет включена в ядро как модуль. После сборки ядра и модулей все опции, скомпилированные в режиме M, будут "лежать" на диске, пока не понадобятся ядру. А как только это произойдет, будет загружен нужный модуль. Вообще, о модулях мы уже говорили, поэтому не будем повторяться.

При выходе из конфигуратора ядра появится вопрос: хотите ли вы сохранить изменения в конфигурации ядра (рис. П4)? Конечно, хотим!

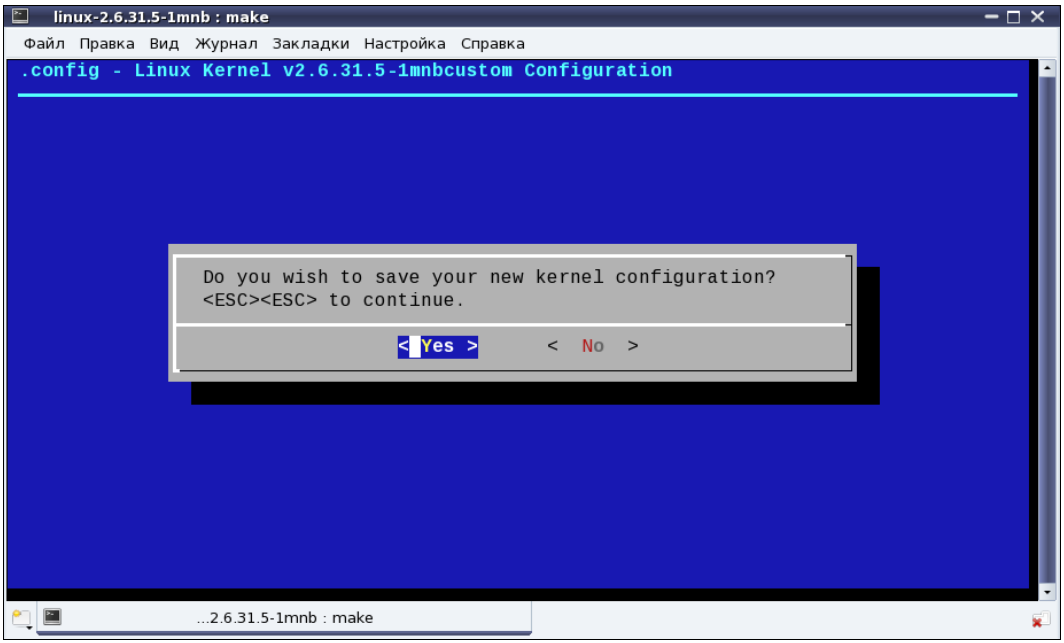


Рис. П4. Сохранить изменения?

П3. Компиляция ядра

После настройки ядра конфигуратор сообщит, что для построения ядра нужно ввести команду `make` (рис. П5), а для вывода справки — `make help`.

Спешить с вводом `make` не будем — ее ввести мы успеем всегда. Лучше введите `make help` для ознакомления с параметрами команды `make`. Если вы внимательно прочитаете вывод команды `make help`, то узнаете, что команда `make` (без параметров) аналогична команде `make all`, которая соберет следующие цели (отмеченные *):

- `vmlinuz` — собирает обычное, "большое" ядро;
- `modules` — собирает модули ядра;
- `bzImage` — собирает сжатое ядро, которое помещается в каталог arch/i386/boot.

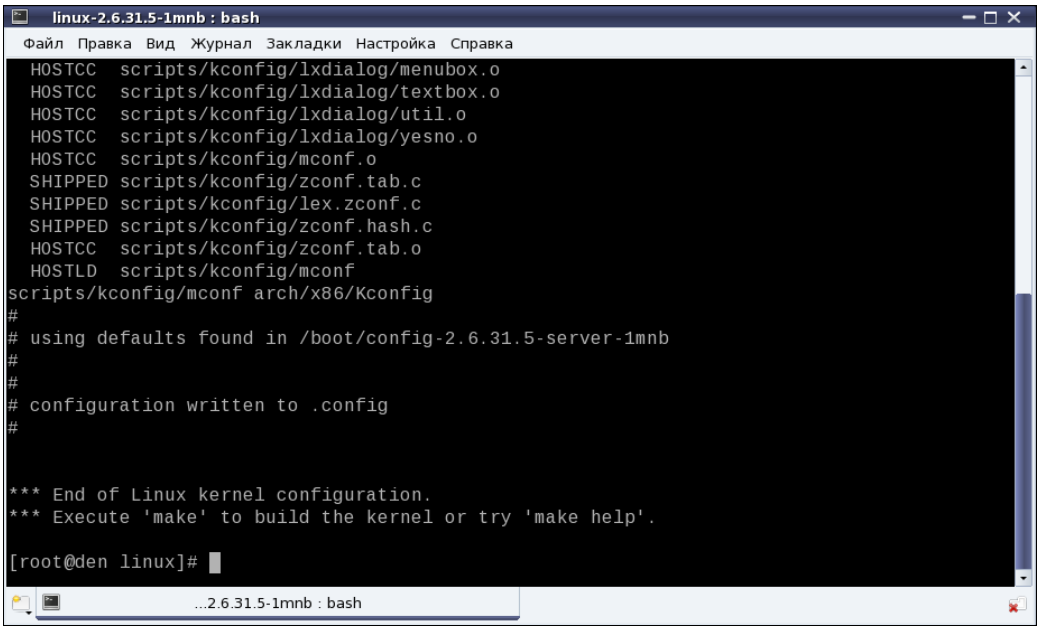


Рис. П5. Введите команду `make`

Теперь понятно, что мы можем просто ввести команду `make` — она проделает как раз то, что нам нужно: `# make`.

> ***Примечание***
>
> Компиляция ядра (рис. П6) — довольно утомительный процесс. Нет, вам ничего делать не нужно, но именно это и утомляет, поэтому можете смело отойти от компьютера и выпить чашечку чая или кофе — не волнуйтесь, вы успеете перекусить к моменту завершения сборки ядра. На среднем компьютере время выполнения этой команды составляет 1–2 часа (этого хватит, чтобы и в магазин сходить, а не только кофе выпить). Один раз я рискнул запустить эту команду в эмуляторе (в виртуальной машине VMware) — на выполнение операции понадобилось 5 часов.

Но это еще не все. Мы только скомпилировали модули и ядро, но пока не устанавливали их. Для установки модулей введите команду:

```
# make modules_install
```

К счастью, данная операция занимает намного меньше времени.

После установки модулей следует установить ядро. Это можно сделать с помощью команды:

```
# make install
```

В процессе установки (рис. П7) будет добавлена соответствующая метка в файл конфигурации загрузчика. После этого следует ввести команду `reboot` для перезагрузки.

При перезагрузке нужно выбрать метку custom_версия_ядра, например **Linux с ядром 2.6.31.5-1mnbcustom** (рис. П1.8). Как только система загрузится, убедитесь, что все нормально работает.

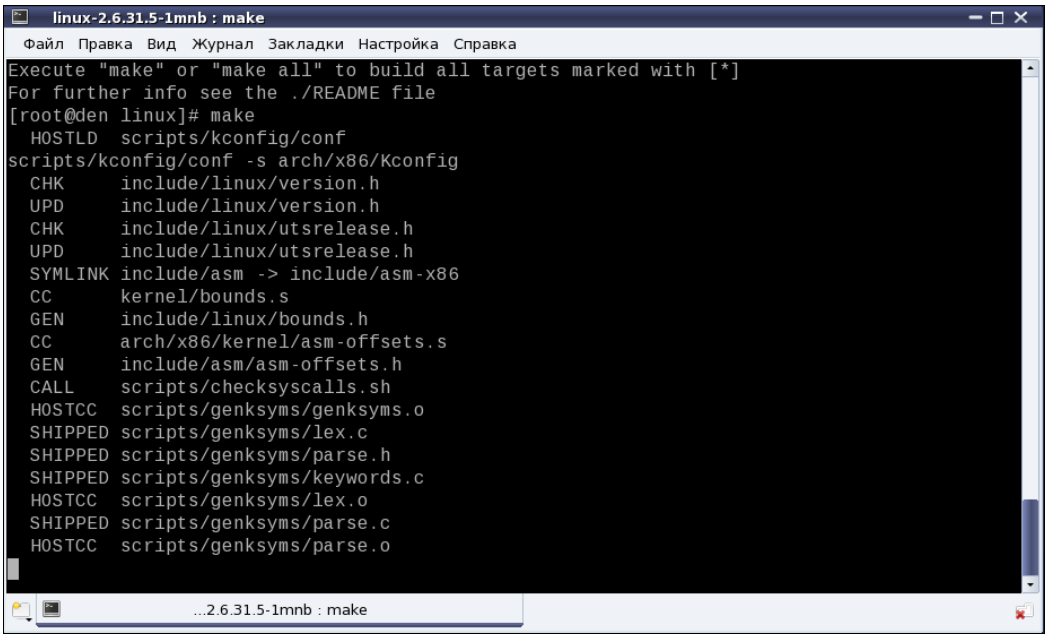


Рис. П6. Компиляция ядра

```
  INSTALL /lib/firmware/emi26/loader.fw
  INSTALL /lib/firmware/emi26/firmware.fw
  INSTALL /lib/firmware/emi26/bitstream.fw
  INSTALL /lib/firmware/emi62/loader.fw
  INSTALL /lib/firmware/emi62/bitstream.fw
  INSTALL /lib/firmware/emi62/spdif.fw
  INSTALL /lib/firmware/emi62/midi.fw
  INSTALL /lib/firmware/kaweth/new_code.bin
  INSTALL /lib/firmware/kaweth/trigger_code.bin
  INSTALL /lib/firmware/kaweth/new_code_fix.bin
  INSTALL /lib/firmware/kaweth/trigger_code_fix.bin
  INSTALL /lib/firmware/ti_3410.fw
  INSTALL /lib/firmware/ti_5052.fw
  INSTALL /lib/firmware/mts_cdma.fw
  INSTALL /lib/firmware/mts_gsm.fw
  INSTALL /lib/firmware/mts_edge.fw
  INSTALL /lib/firmware/edgeport/boot.fw
  INSTALL /lib/firmware/edgeport/boot2.fw
  INSTALL /lib/firmware/edgeport/down.fw
  INSTALL /lib/firmware/edgeport/down2.fw
  INSTALL /lib/firmware/edgeport/down3.bin
  INSTALL /lib/firmware/whiteheat_loader.fw
  INSTALL /lib/firmware/whiteheat.fw
  INSTALL /lib/firmware/keyspan_pda/keyspan_pda.fw
  INSTALL /lib/firmware/keyspan_pda/xircom_pgs.fw
  INSTALL /lib/firmware/vicam/firmware.fw
  INSTALL /lib/firmware/cpia2/stv0672_vp4.bin
  INSTALL /lib/firmware/yam/1200.bin
  INSTALL /lib/firmware/yam/9600.bin
  DEPMOD  2.6.31.5-1mnbcustom

[root@den linux]#
[root@den linux]# make install
sh /usr/src/linux-2.6.31.5-1mnb/arch/x86/boot/install.sh 2.6.31.5-1mnbcustom arch/x86/boot/bzImage \
              System.map "/boot"
defaulting background resolution to 1024x768
[root@den linux]#
```

Рис. П7. Установка модулей и ядра (`make install`)

Рис. П8. Выбор нового ядра при загрузке

В процессе компиляции ядра создается много ненужных после его завершения файлов (у меня такого "мусора" насобиралось на 3 Гбайт), а команда `make clean` позволяет весь этот "мусор" удалить. Так что в заключение произведем "генеральную уборку" — откройте терминал и от имени root введите команды:

```
# cd /usr/src/linux
# make clean
```

Поздравляю! Вы успешно справились с перекомпиляцией ядра.

Предметный указатель

A

Almquist shell 19
APM 418
ar 73
ash 19
AT&T 16
autoconf 66
automake 66

B

bash 14, 17, 20
binutils 61
breakpoint 396

C

csh 16

D

dialog 45
DNS 245
DOS 196

E

Entry point 112

F

FIFO 137
File mode 90
FreeBSD 16
FTP 245

G

gcc 61
◊ -fPIC 75
gcc-c++ 61
gdb 395
Genie 413
GID 233
Glade 382
gprof 407
GTK+ 338
GTK-сигналы 355

H

HTTP 245

I

IMAP 245
init 79
Input/Output 83
IP 243
IPC 123, 133
IPC Key 140
IPng 246
IPv6 246
IP-адрес 245
i-узел 200

K

kbuild 165
Key loggers 187
klogd 163
ks 16

L

Lazarus 382
Login shell 32

M

MAC-адрес 243
make 61, 161
Makefile 66
Mandriva 206
MBR 199
mcedit 60
Monodevelop 414
Multics 109

N

nano 60
NAT 246
NIC 246
nmap 332

O

OSI 242

P

pdksh 17
Permissions 90
PID 100, 120
POP 245
Position Independent Code 75
POSIX 16
PPID 120
proc 234

Q

QtDesigner 382
QtDevelop 382

R

RAD 382

S

Segmentation fault 405
sh 15
SMP 111
SMTP 245
SSL 244
sysctl 142
sysfs 234
syslog 163
syslog-ng 163
System V 140

T

tar 73
TCL 266
TCL-команда
◊ append 285
◊ array 289
◊ bind 319
◊ button 304
◊ checkbutton 307
◊ concat 284
◊ entry 315
◊ exec 333
◊ expr 275
◊ foreach 288
◊ format 272
◊ gets 274, 293
◊ grid 321
◊ if 276
◊ incr 277
◊ join 288
◊ lappend 284
◊ lindex 284
◊ linsert 285
◊ list 283
◊ listbox 318
◊ lrange 284
◊ lreplace 285
◊ lsearch 285
◊ menu 313
◊ message 329
◊ pack 268, 302
◊ puts 269, 271
◊ radiobutton 311
◊ read 295
◊ seek 296
◊ set 272
◊ spinbox 315
◊ split 287
◊ string 278
◊ tell 296

◊ text 333
◊ tk_getOpenFile 330
◊ tk_getSaveFile 330
◊ tk_mesagebox 328
◊ toplevel 328
◊ while 277
TCP 244
TCP/IP 244
tcsh 18, 31
TENEX 18, 31
THREAD_ID 123
Threads 122
TK 266

U

Ubuntu 31, 208
udev 202
UID 120, 233
UUID 202

V

Vala 413
vi 60
Virtual File System 234

X

Xdialog 57

Z

zsh 17

Б

Библиотека
◊ GDK 339
◊ glib 339
◊ Makefile 76
◊ pthread 123
◊ динамическая 71
◊ статическая 71
Блок 199

В

Виджет 298, 346
Выражение 35

Г

Главная загрузочная запись 199

Д

Дескриптор 249
Диск
◊ гибкий 202

Ж

Журналы 194

З

Заголовок
◊ /usr/src/linux/include/linux/msg.h 141
◊ /usr/src/linux/include/linux/sem.h 149
◊ /usr/src/linux/include/linux/shm.h 154
◊ dirent.h 215, 216
◊ fcntl.h 92, 93
◊ glib.h 339
◊ glist.h 342
◊ gslist.h 342
◊ gtimer.h 344
◊ gtk.h 347
◊ gtk/gtkwidget.h 353, 361, 362
◊ gtk/gtkwindow.h 353
◊ kernel.h 162
◊ linux/fs.h 175
◊ linux/init.h 164
◊ linux/ipc.h 143
◊ locale.h 360
◊ mntent.h 226

Заголовок *(прод.)*
◊ module.h 162
◊ netinet/in.h 250
◊ proc_fs.h 184
◊ pthread.h 129
◊ pwd.h 121
◊ signal.h 120
◊ socket.h 249
◊ stdarg.h 86
◊ stdio.h 87, 223
◊ stdlib.h 81, 107
◊ strings.h 229
◊ sys/stat.h 93, 217
◊ sys/types.h 93, 121
◊ unistd.h 80, 93, 95, 113, 215
◊ wait.h 118

К

Кадр 243
Карта блоков 200
Каталог
◊ /etc/skel 33
◊ домашний 213
◊ права доступа 230
◊ признак каталога 230
◊ родительский 213
◊ текущий 213
Квант 112
Команда
◊ alias 33
◊ builtins 31
◊ cat 220
◊ cd 213
◊ chmod 231
◊ chmod +x 23
◊ chown 233
◊ cp 220
◊ fdisk 203
◊ fsck 206
◊ gtk-config 349
◊ insmod 159
◊ ipcs 141
◊ kill 102
◊ killall 104
◊ ldconfig 76
◊ less 220
◊ limit 89
◊ llength 284
◊ ln 222
◊ locate 220
◊ ls 213
◊ mkdir 213
◊ mknod 137, 175
◊ modinfo 169
◊ modprobe 160
◊ mv 220
◊ nice 107
◊ ps 100, 102
◊ pstree 100
◊ renice 107
◊ rm 213, 220
◊ rmdir 213
◊ rmmod 162
◊ setenv 33
◊ strace 404
◊ tac 220
◊ top 105
◊ touch 220
◊ ulimit 89
◊ umount 202
◊ which 220

М

Маска сети 247
Массивы 25

Н

Нить 111

О

Оператор 35
◊ case 28
◊ if 27
Очередь заданий 188

П

Пакет
◊ kernel-source 415
Переменная окружения 34
◊ HOME 79
◊ LD_LIBRARY_PATH 75
◊ PATH 79
◊ PWD 79
◊ SHELL 79
◊ USER 79

Переменные 24
◊ окружения 24
◊ специальные 25
Протокол 244

Р

Рефал 413

С

Семафоры 149
Сигнал 119
Системный вызов
◊ chdir() 215
◊ chmod() 232
◊ chown() 233
◊ close() 89, 249
◊ creat() 89, 92
◊ execl() 112, 116
◊ execle() 116
◊ execlp() 116
◊ execve() 114, 115
◊ execvp() 116
◊ fork() 112, 122
◊ getcwd() 229
◊ getgid() 233
◊ getuid() 233
◊ link() 224
◊ lseek() 89, 97
◊ lstat() 217
◊ mkdir() 219
◊ mknod() 137
◊ mount() 210
◊ msgctl() 148
◊ msgget() 143
◊ msgrcv() 147
◊ msgsend() 145
◊ open() 89, 94
◊ opendir() 215
◊ read() 89, 94
◊ readdir() 216
◊ readlink() 224
◊ readv() 89
◊ rename() 223
◊ rewinddir() 217
◊ rmdir() 219
◊ semctl() 152
◊ semget() 150
◊ semop() 151
◊ shmat() 155
◊ shmctl() 156
◊ shmdt() 156
◊ shmget() 154
◊ signal() 119
◊ statvfs() 227
◊ symlink() 224
◊ umask() 138, 224
◊ unlink() 223
◊ wait() 117
◊ write() 89, 95
◊ writev() 89
События 298, 355
Сокет 249
Структура
◊ dirent 216
◊ file_operations 175
◊ ipc_perm 143
◊ mntent 226
◊ msgbuf 141
◊ msqid_ds 142
◊ proc_dir_entry 184
◊ sembuf 151
◊ semid_ds 149
◊ shmid_ds 154
◊ sockadd_in 251
◊ sockadd_in6 251
◊ sockaddr 250
◊ stat 217
◊ statvfs 227
Суперблок 200
Сценарий 15

Т

Точка монтирования 207

Ф

Файл
◊ .bash_history 20
◊ /etc/csh.cshrc 32
◊ /etc/csh.login 32
◊ /etc/csh.logout 32
◊ /etc/fstab 205, 206, 226
◊ /etc/ld.so.cache 75
◊ /etc/ld.so.conf 75
◊ /etc/modprobe.conf 160

Файл *(прод.)*
◊ /etc/modprobe.preload 161
◊ /etc/modules.conf 159
◊ /etc/mtab 226
◊ /etc/profile 20
◊ /etc/protocols 250
◊ /etc/resolv.conf 363
◊ /etc/shells 15
◊ /etc/sysctl.conf 239
◊ /proc/filesystems 234
◊ /proc/kallsyms 172
◊ /proc/mounts 226
◊ ~/.bash_logout 20
◊ ~/.bash_profile 20
◊ ~/.bashrc 20
◊ ~/.history 33
◊ Documentation/devices.txt 177
◊ права доступа 230
◊ устройства 202
Файловая система 194, 195, 198
◊ ext2 194
◊ ext3 194
◊ JFS 195
◊ ReiserFS 195
◊ XFS 195
◊ журналируемая 194
Фрейм 326
Функция
◊ accept() 253
◊ basename() 229
◊ bind() 249, 250
◊ cleanup_module() 162
◊ clearenv() 82
◊ connect() 249, 252, 253
◊ endmntent() 227
◊ fchdir() 215
◊ fclose() 85
◊ fflush() 87
◊ fgetpos() 87
◊ fgets() 86
◊ fopen() 85, 226
◊ fopendir() 216
◊ fprintf() 86
◊ fputs() 86
◊ freopen() 85
◊ fscanf() 86
◊ fseek() 86
◊ fsetpos() 87
◊ ftell() 87
◊ ftok() 140
◊ g_object_ref() 361
◊ get_user() 183
◊ getenv() 80
◊ gethostbyname() 254
◊ getmntent() 226
◊ getpid() 132
◊ getpwuid() 121
◊ getsockname() 254
◊ getsockopt() 261
◊ gtk_check_button_new() 370
◊ gtk_check_button_new_with_label() 370
◊ gtk_clist_append() 375
◊ gtk_clist_clear() 375
◊ gtk_clist_insert() 375
◊ gtk_clist_new() 375
◊ gtk_clist_new_with_titles() 375
◊ gtk_clist_prepend() 375
◊ gtk_clist_remove() 375
◊ gtk_container_remove() 361
◊ gtk_entry_new() 364
◊ gtk_entry_set_editable() 364
◊ gtk_init() 348
◊ gtk_main() 348
◊ gtk_main_quit() 351
◊ gtk_radio_button_group() 371
◊ gtk_radio_button_new() 370
◊ gtk_radio_button_new_with_label() 370
◊ gtk_signal_connect() 351, 354
◊ gtk_toggle_button_get_active() 371
◊ gtk_toggle_button_set_active() 371
◊ gtk_widget_destroy() 361
◊ gtk_widget_grab_focus() 362
◊ gtk_widget_hide() 362
◊ gtk_widget_show() 362
◊ gtk_window_new() 348
◊ gtk_window_set_default_size() 352
◊ gtk_window_set_title() 348
◊ init_module() 162
◊ ioctl() 183, 264
◊ listen() 253
◊ msg_exists() 148
◊ pclose() 134
◊ popen() 134
◊ printk() 162
◊ pthread() 125
◊ pthread_cancel() 132
◊ pthread_create() 123
◊ pthread_exit() 129

◊ pthread_join() 130
◊ pthread_self() 132
◊ put_user() 180
◊ putenv() 81
◊ recv() 254
◊ recvfrom() 254
◊ register_chrdev() 177
◊ rewind() 87
◊ send() 254
◊ sendto() 254
◊ setenv() 81
◊ setlocale() 360
◊ setmntent() 226
◊ setsockopt() 261
◊ shutdown() 249, 255
◊ socket() 249, 250
◊ system() 107
◊ unsetenv() 81
◊ vfscanf() 86
◊ vfsprintf() 86

Ц

Целевые связки 66
Цели Makefile 66
Цикл
◊ for 26
◊ while 26

Я

Ядро
◊ конфигуратор 417
◊ модуль ядра 419